Lecture 6

Red-Black Trees: Insertion, Deletion

Source: Introduction to Algorithms, CLRS

RB-Trees: Insertion Cases

RB-Trees: Insertion Cases

Let z be the newly inserted node with colour red. Then,

RB-Trees: Insertion Cases

Let z be the newly inserted node with colour red. Then,

- If parent of z is **black**, nothing needs to be done.

RB-Trees: Insertion Cases

Let z be the newly inserted node with colour red. Then,

- If parent of z is **black**, nothing needs to be done.
- If parent of z is red, we do fix-ups for the following cases:

RB-Trees: Insertion Cases

Let z be the newly inserted node with colour red. Then,

- If parent of z is **black**, nothing needs to be done.
- If parent of z is red, we do fix-ups for the following cases:
    - **Case 1:** z's **uncle** (sibling of z's parent) is red.

RB-Trees: Insertion Cases

Let z be the newly inserted node with colour red. Then,

- If parent of z is **black**, nothing needs to be done.
- If parent of z is red, we do fix-ups for the following cases:
  - **Case 1:** z's **uncle** (sibling of z's parent) is red.
  - **Case 2:** z's **uncle** is **black** and z is a **right child**.

RB-Trees: Insertion Cases

Let z be the newly inserted node with colour red. Then,

- If parent of z is **black**, nothing needs to be done.
- If parent of z is red, we do fix-ups for the following cases:
    - **Case 1:** z's **uncle** (sibling of z's parent) is red.
    - **Case 2:** z's **uncle** is **black** and z is a **right child**.
    - **Case 3:** z's **uncle** is **black** and z is a **left child**.

RB-Trees: Insertion Cases

Let z be the newly inserted node with colour red. Then,

- If parent of z is **black**, nothing needs to be done.
- If parent of z is red, we do fix-ups for the following cases:
  - **Case** 1**:** z's **uncle** (sibling of z's parent) is red.
  - **Case** 2**:** z's **uncle** is **black** and z is a **right child**.
  - **Case** 3**:** z's **uncle** is **black** and z is a **left child**.

After doing local fix-up, z will set to its parent's parent.

RB-Trees: Insertion Cases

Let z be the newly inserted node with colour red. Then,

- If parent of z is **black**, nothing needs to be done.
- If parent of z is red, we do fix-ups for the following cases:
  - **Case** 1**:** z's **uncle** (sibling of z's parent) is red.
  - **Case** 2**:** z's **uncle** is **black** and z is a **right child**.
  - **Case** 3**:** z's **uncle** is **black** and z is a **left child**.

After doing local fix-up, z will set to its parent's parent. (Case 1)

Gets converted to Case 3 (Case 2)

RB-Trees: Insertion Cases

Let z be the newly inserted node with colour red. Then,

- If parent of z is **black**, nothing needs to be done.
- If parent of z is red, we do fix-ups for the following cases:
  - **Case** 1**:** z's **uncle** (sibling of z's parent) is red.
  - **Case** 2**:** z's **uncle** is **black** and z is a **right child**.
  - **Case** 3**:** z's **uncle** is **black** and z is a **left child**.

After doing local fix-up, z will set to its parent's parent.

Gets converted to Case 3

Fix-up will be enough to terminate the process

RB-Trees: Insertion Cases

Let z be the newly inserted node with colour red. Then,

- If parent of z is **black**, nothing needs to be done.
- If parent of z is red, we do fix-ups for the following cases:
    - **Case 1:** z's **uncle** (sibling of z's parent) is red.
    - **Case 2:** z's **uncle** is **black** and z is a **right child**.
    - **Case 3:** z's **uncle** is **black** and z is a **left child**.
- If parent of z does not exist, make z **black** and **exit**.

RB-Trees: Insertion Cases

Let z be the newly inserted node with colour red. Then,

- If parent of z is **black**, nothing needs to be done.
- If parent of z is red, we do fix-ups for the following cases:
  - **Case 1:** z's **uncle** (sibling of z's parent) is red.
  - **Case 2:** z's **uncle** is **black** and z is a **right child**.
  - **Case 3:** z's **uncle** is **black** and z is a **left child**.
- If parent of z does not exist, make z **black** and **exit**.

We will see the fix ups assuming parent of z is a left child.

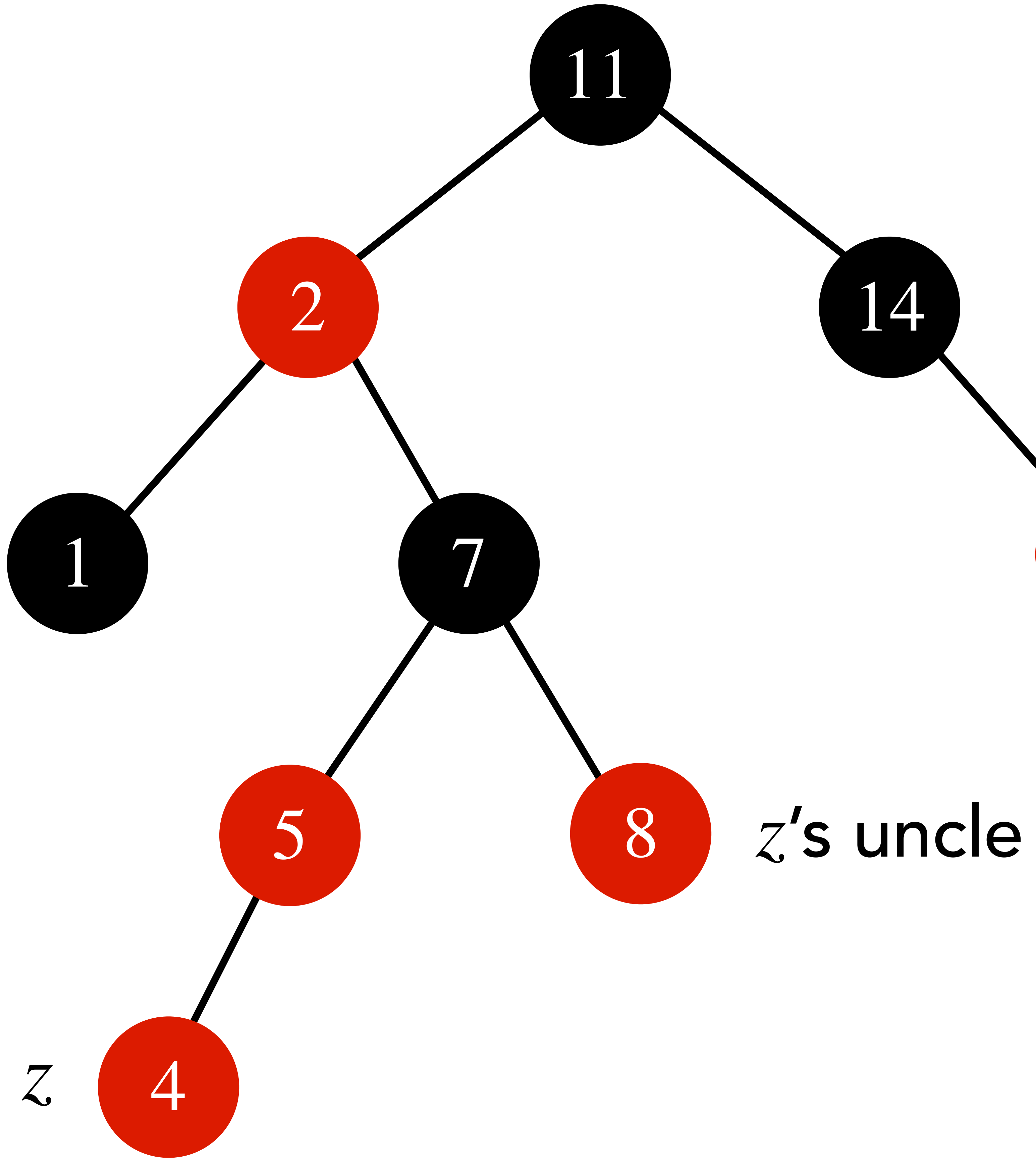
RB-Trees: Insertion Case 1

RB-Trees: Insertion Case 1

Case 1: z's **uncle** (sibling of z's parent) is red.

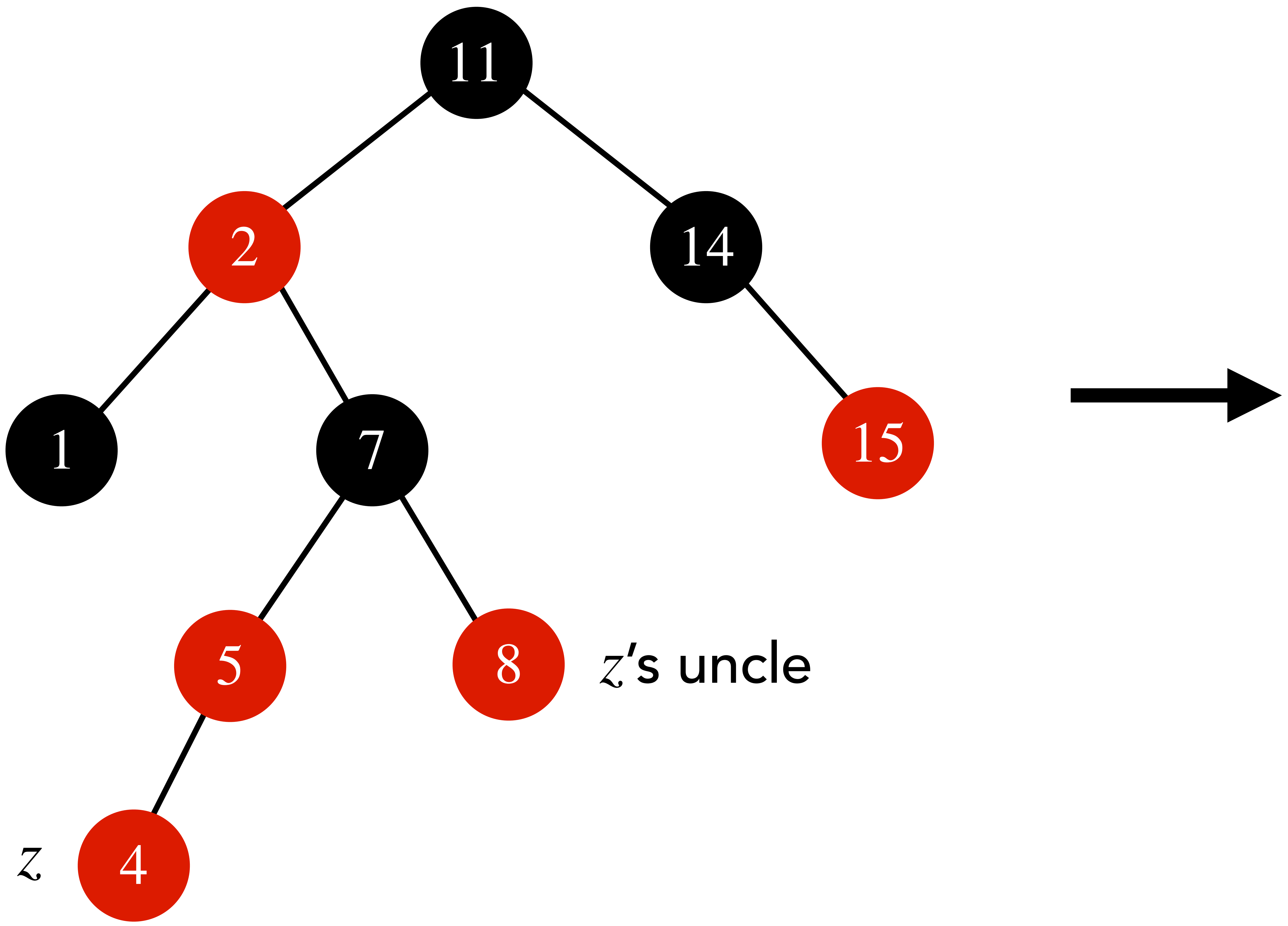
RB-Trees: Insertion Case 1

Case 1**:** z's **uncle** (sibling of z's parent) is red.



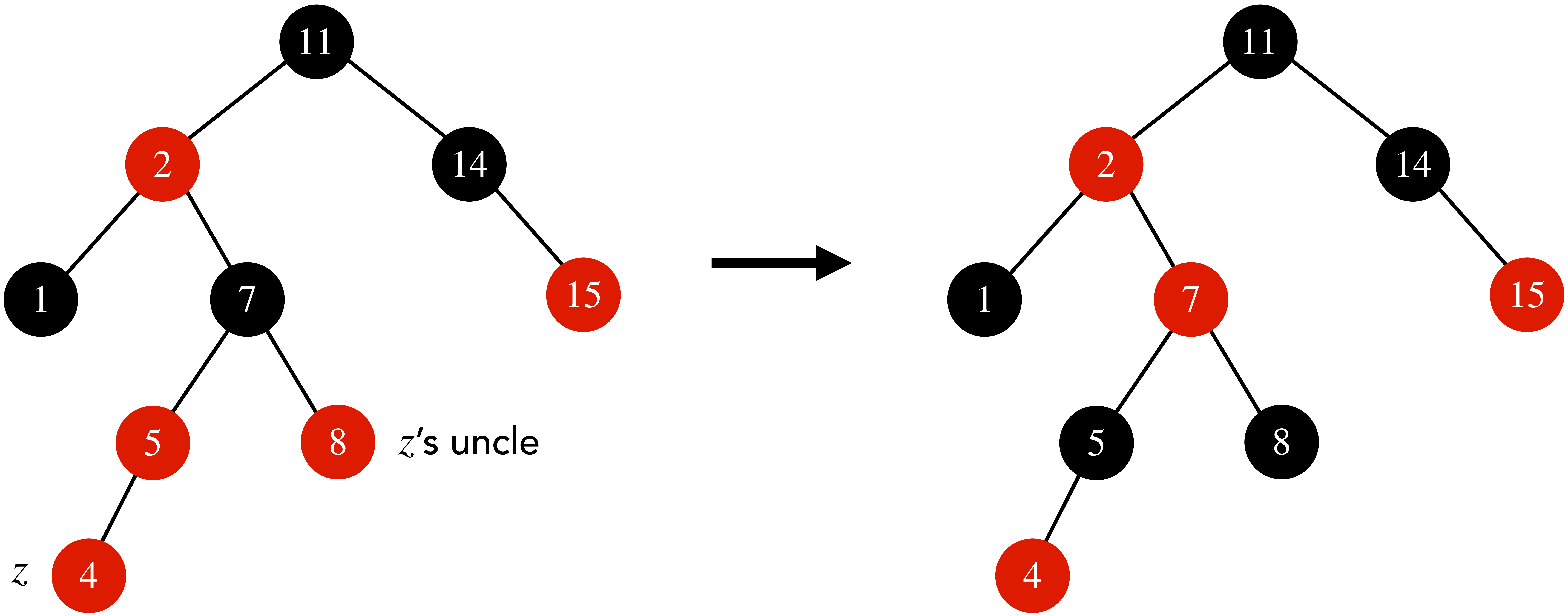
RB-Trees: Insertion Case 1

Case 1**:** z's **uncle** (sibling of z's parent) is red.



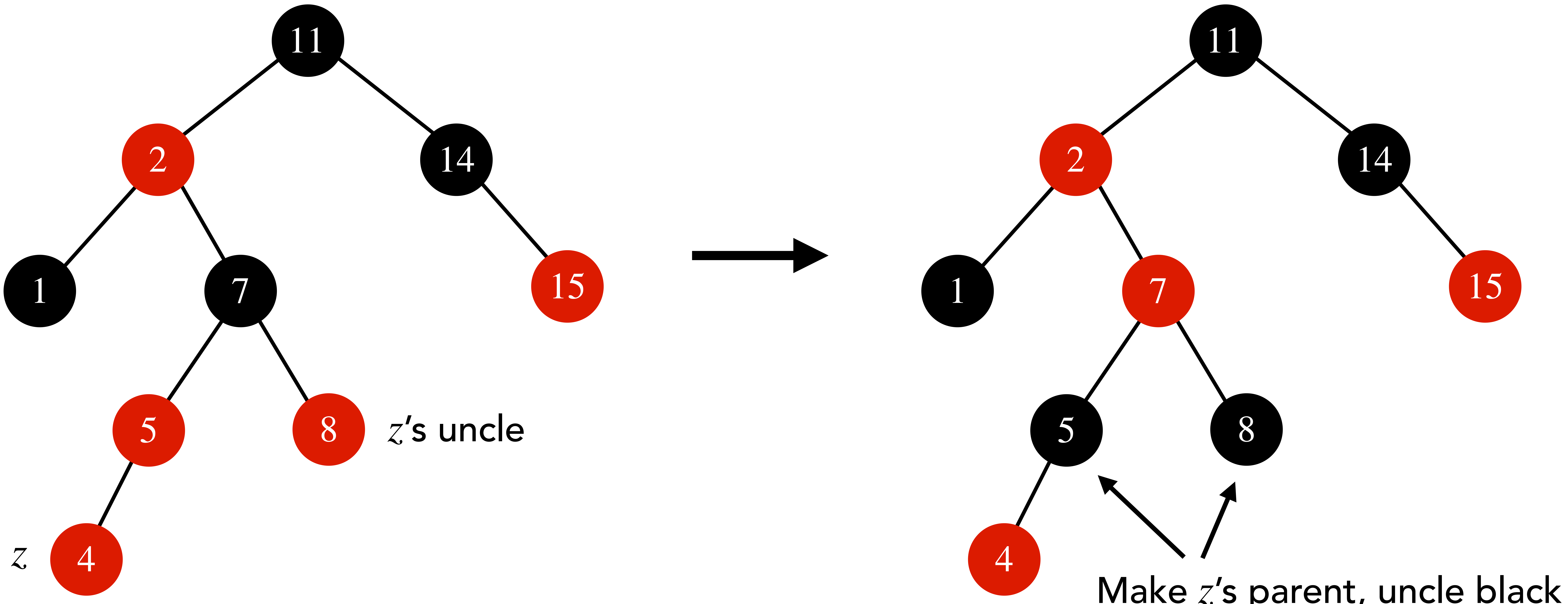
RB-Trees: Insertion Case 1

Case 1**:** z's **uncle** (sibling of z's parent) is red.



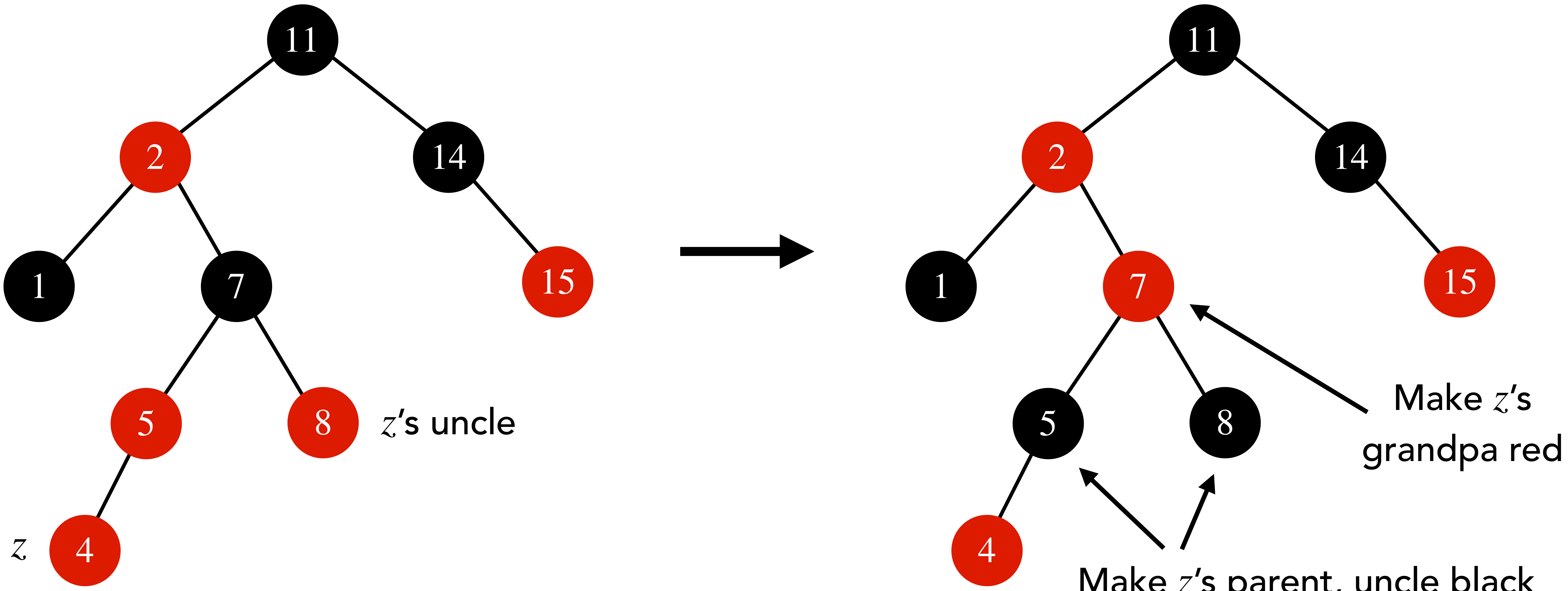
RB-Trees: Insertion Case 1

Case 1: z's **uncle** (sibling of z's parent) is red.



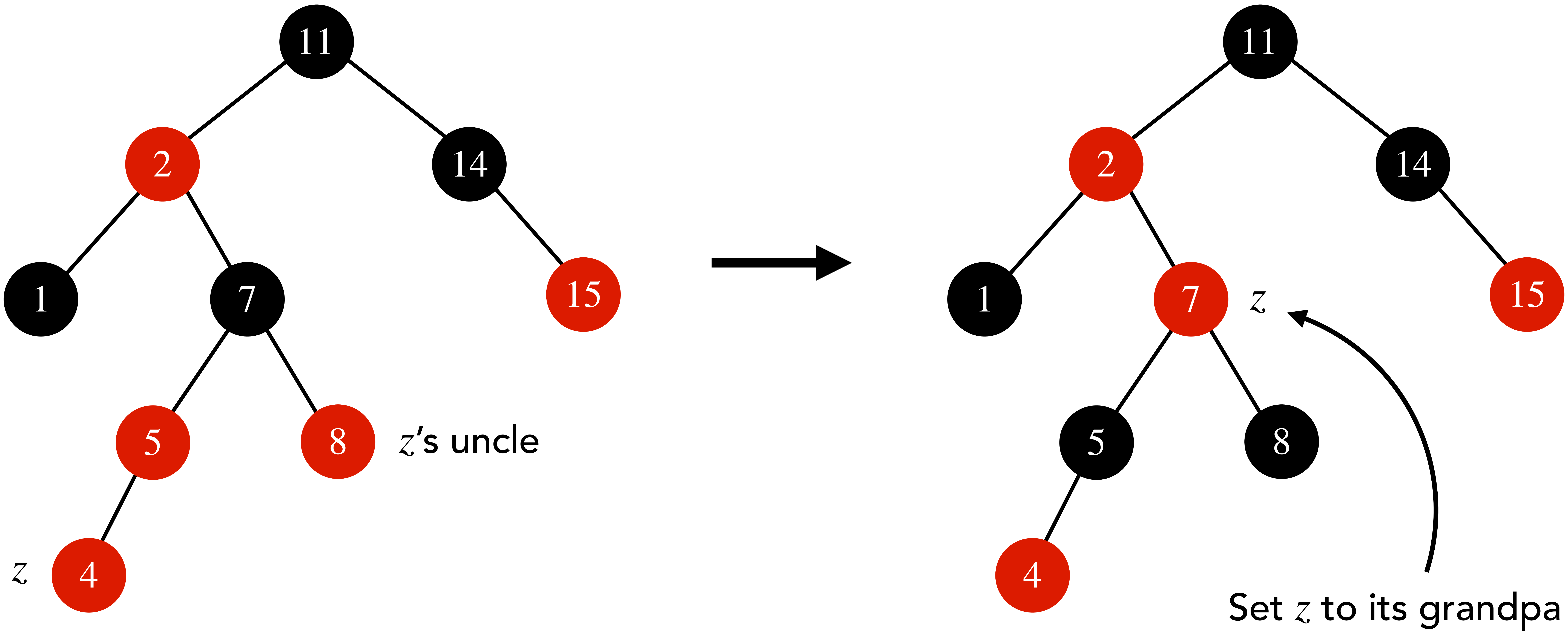
RB-Trees: Insertion Case 1

Case 1: z's uncle (sibling of z's parent) is red.



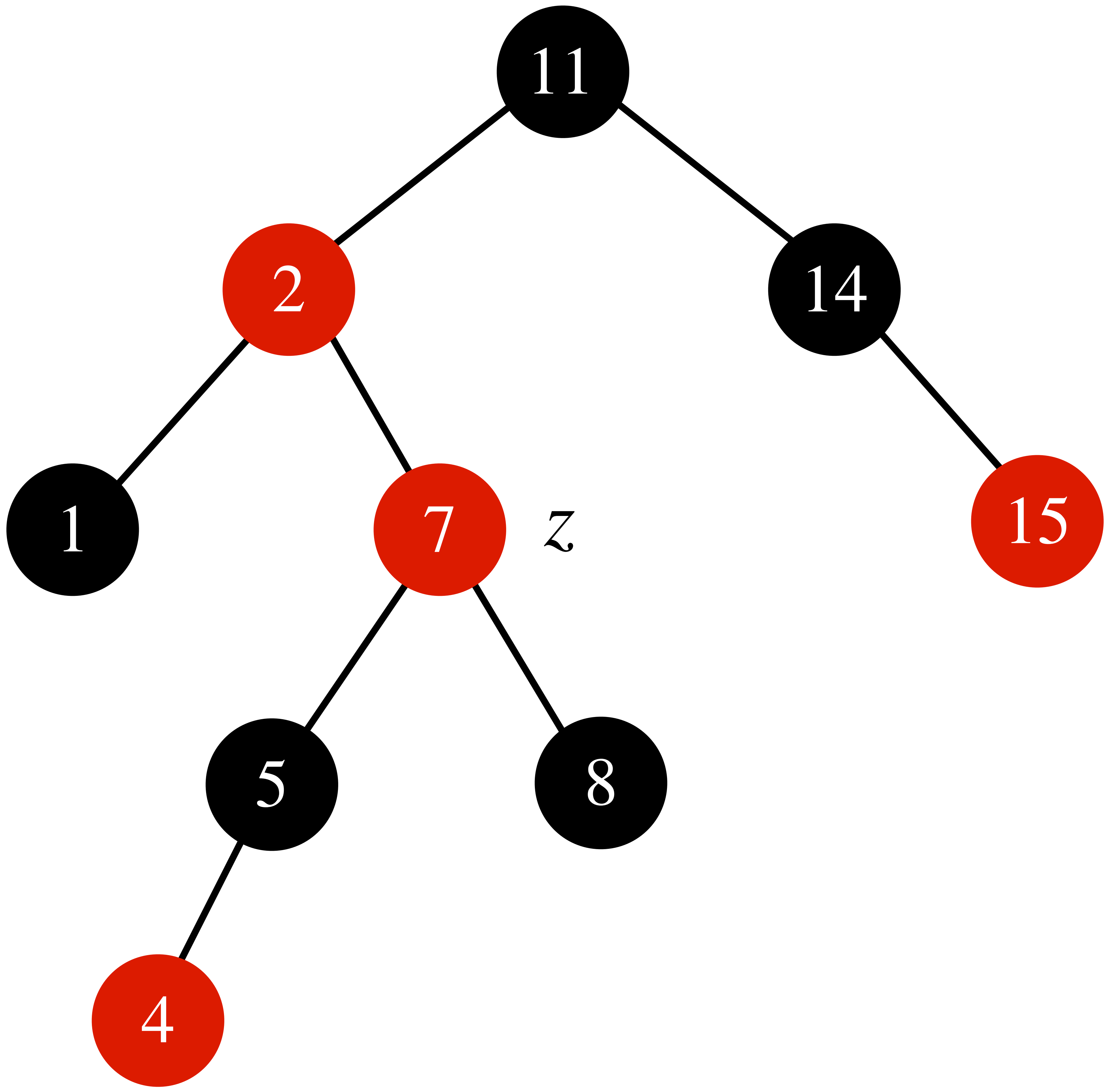
RB-Trees: Insertion Case 1

Case 1**:** z's **uncle** (sibling of z's parent) is red.



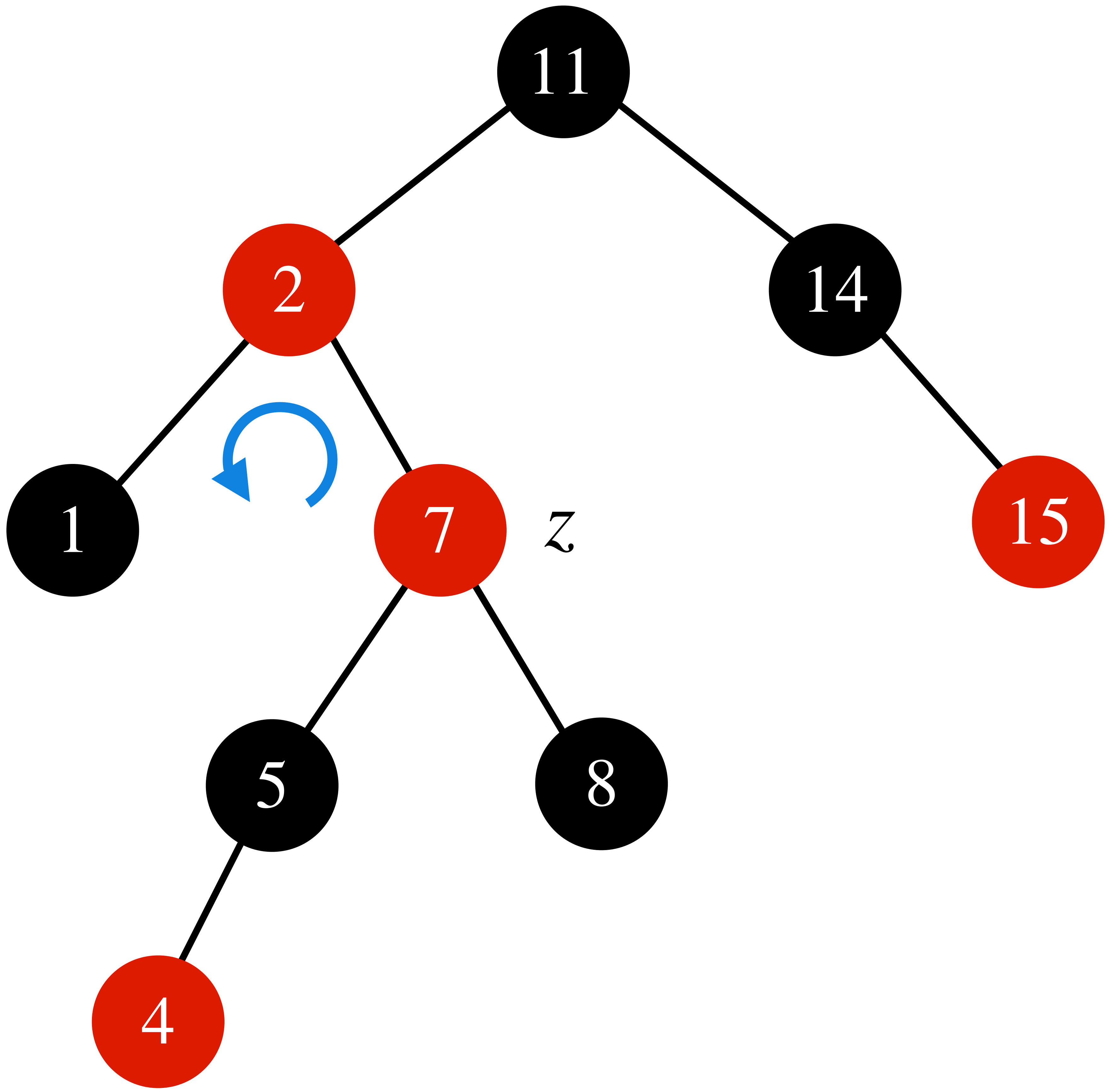
RB-Trees: Insertion Case 2

Case 2: z's uncle is black and z is a right child.



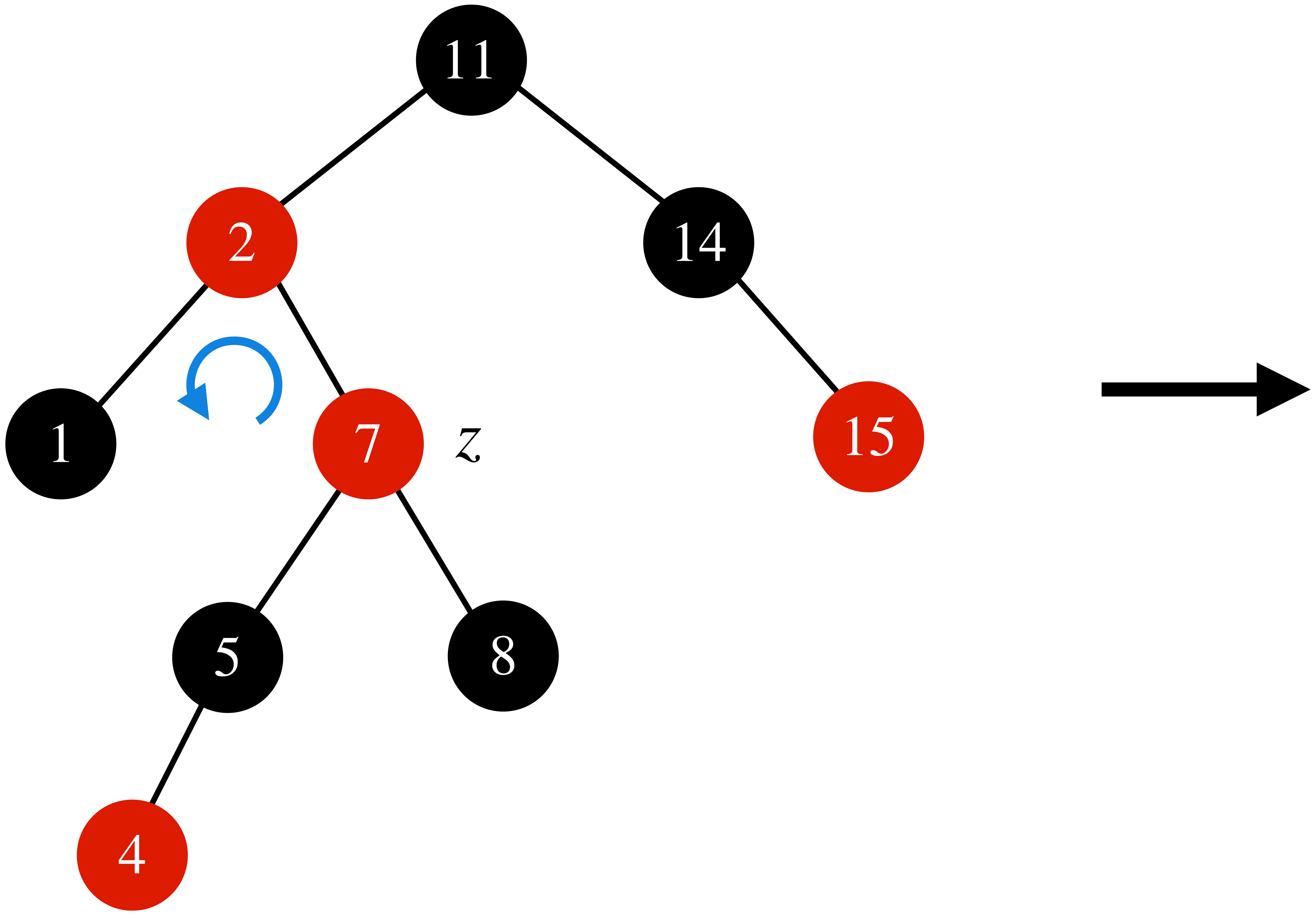
RB-Trees: Insertion Case 2

Case 2**:** z's uncle is black and z is a right child.



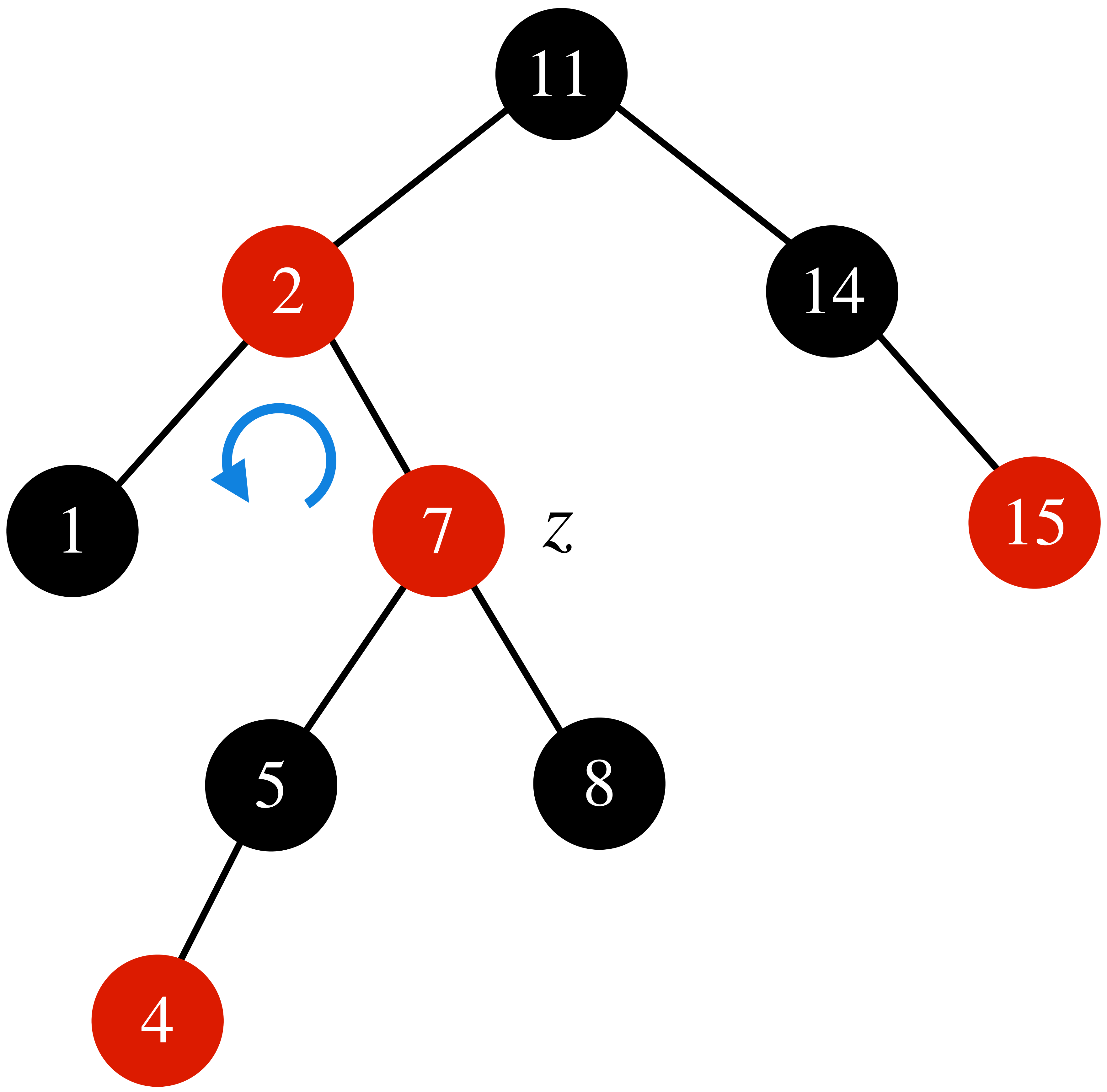
RB-Trees: Insertion Case 2

Case 2: z's uncle is black and z is a right child.



RB-Trees: Insertion Case 2

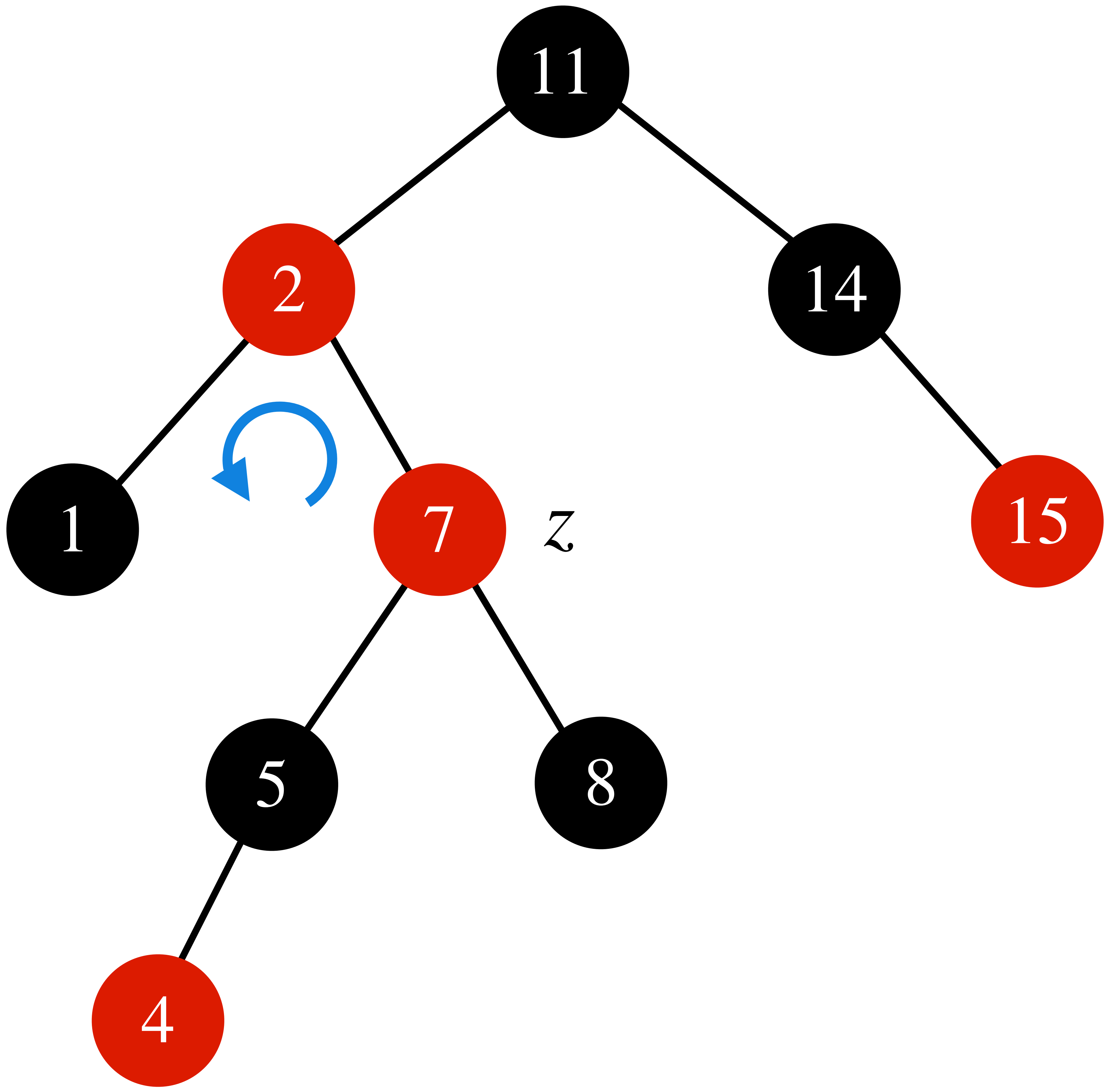
Case 2**:** z's uncle is black and z is a right child.



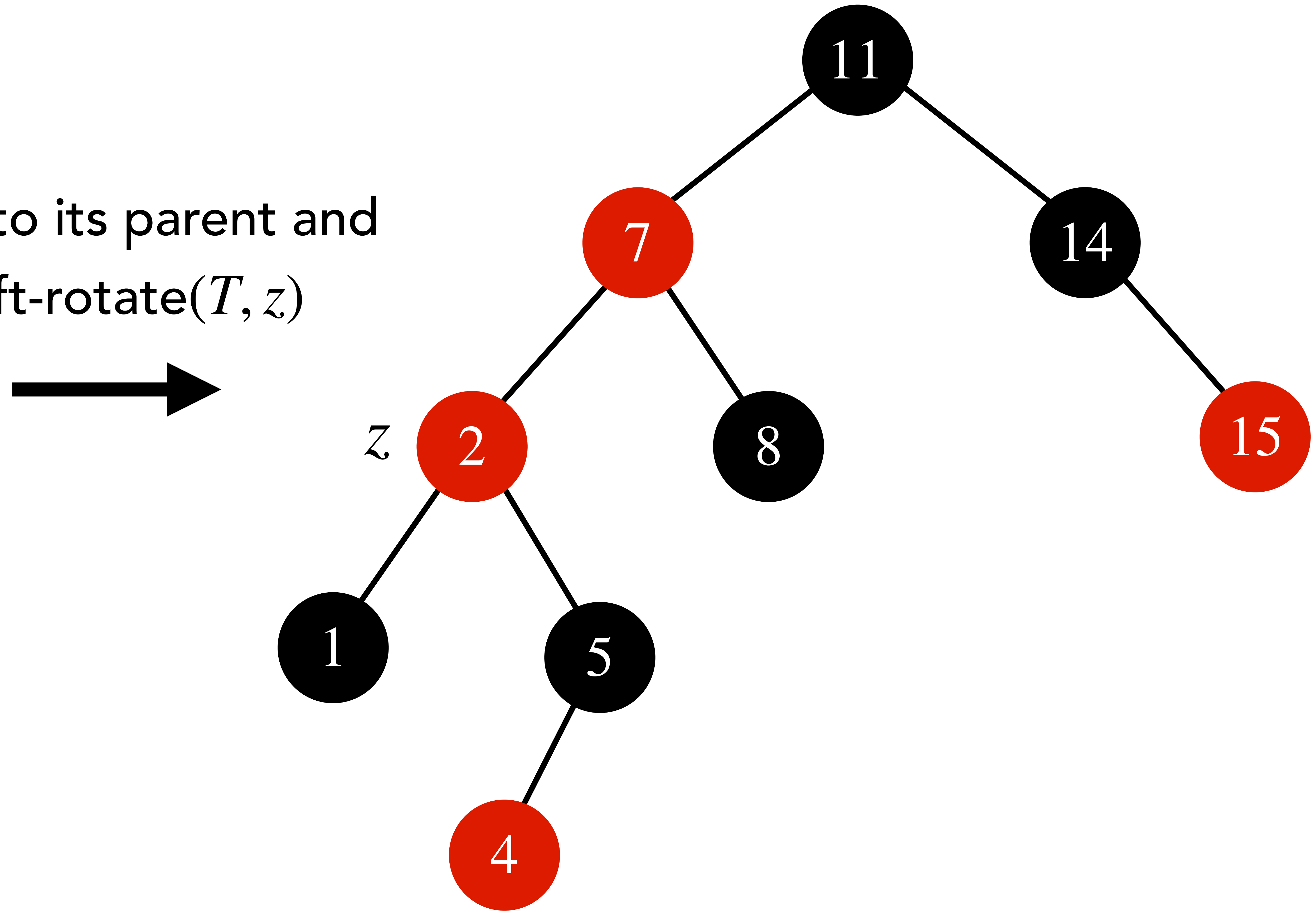
Set z to its parent and Left-rotate(T, z)

RB-Trees: Insertion Case 2

Case 2: z's uncle is black and z is a right child.

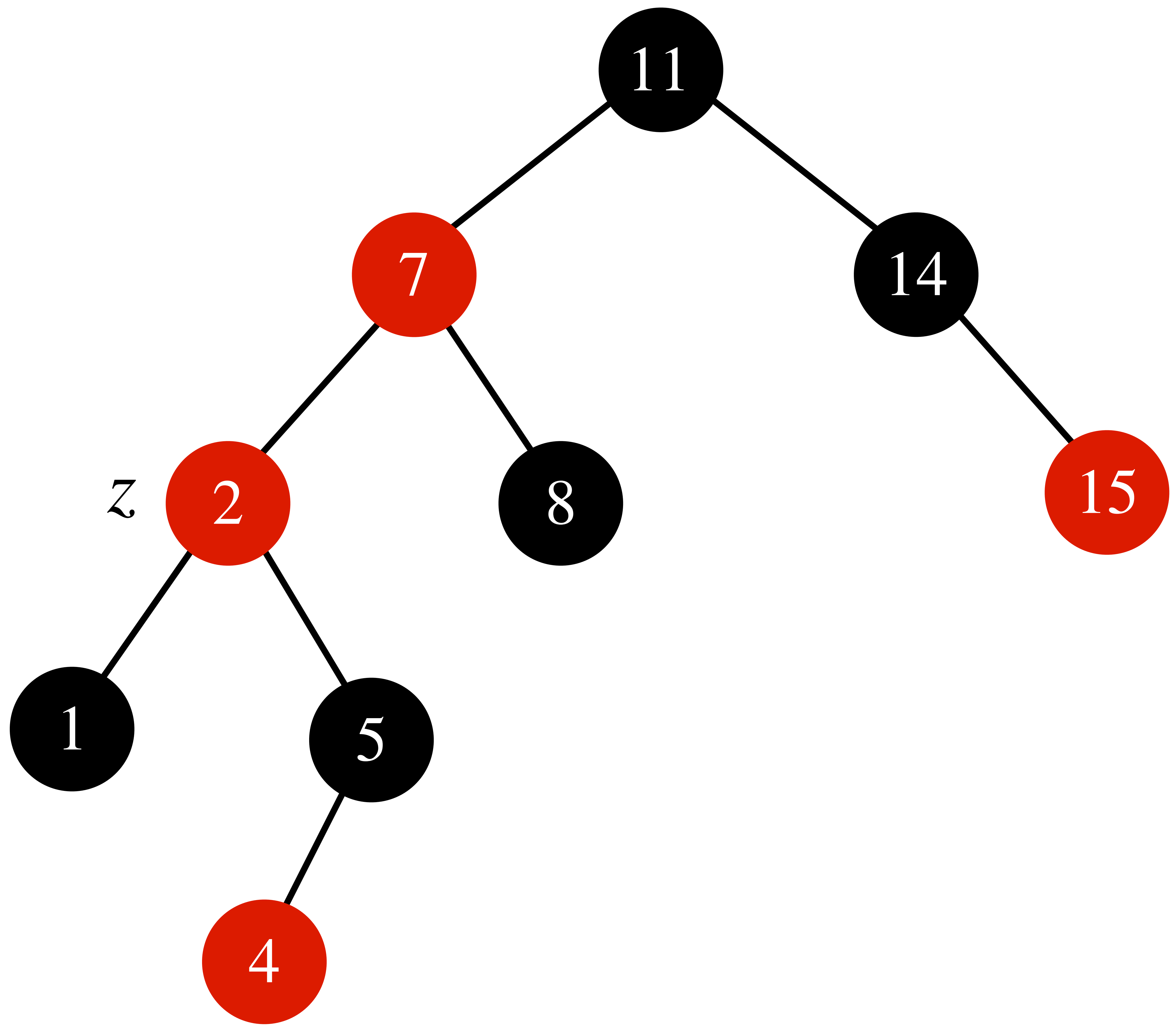


Set z to its parent and Left-rotate(T, z)



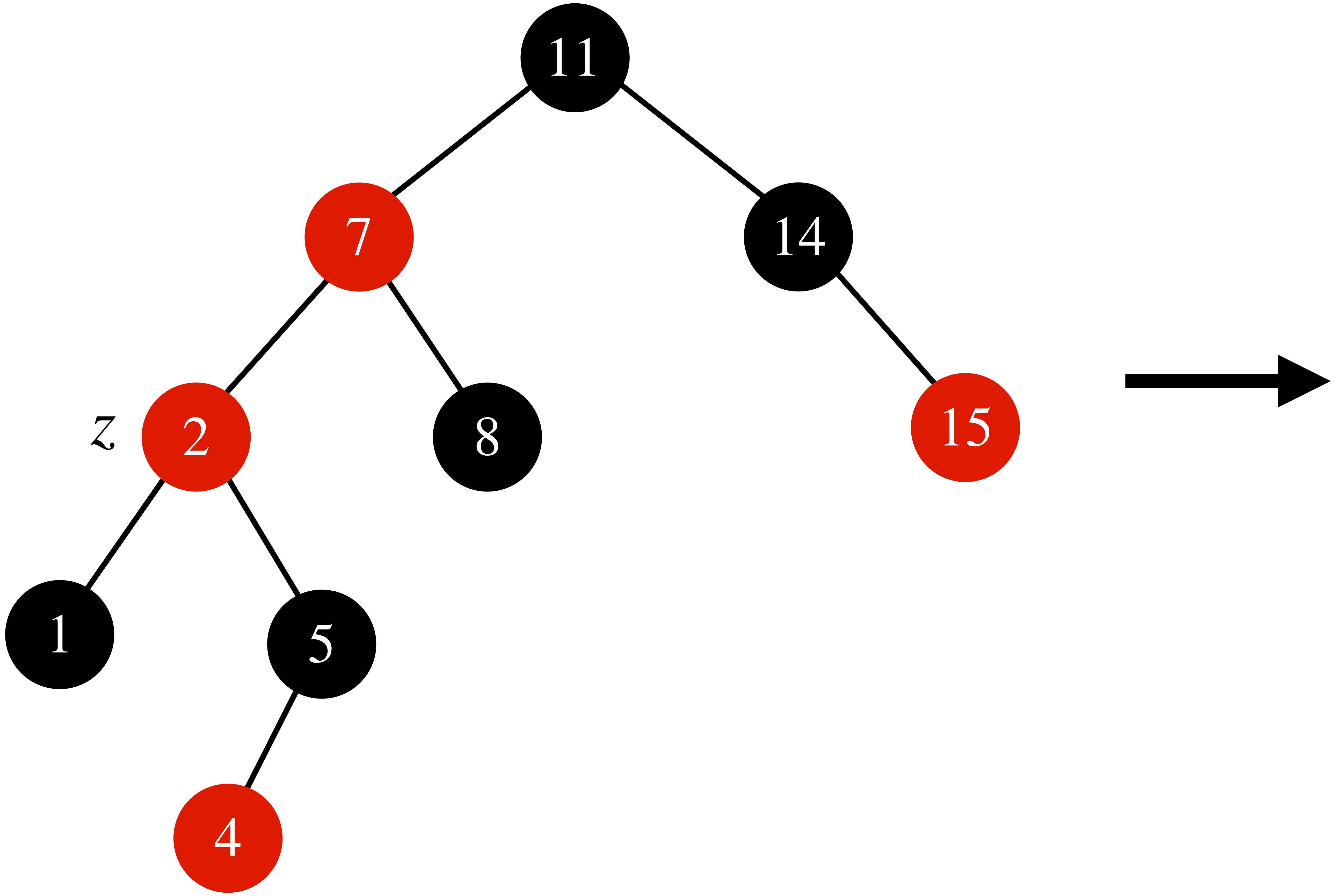
RB-Trees: Insertion Case 3

Case 3: z's uncle is black and z is a left child.



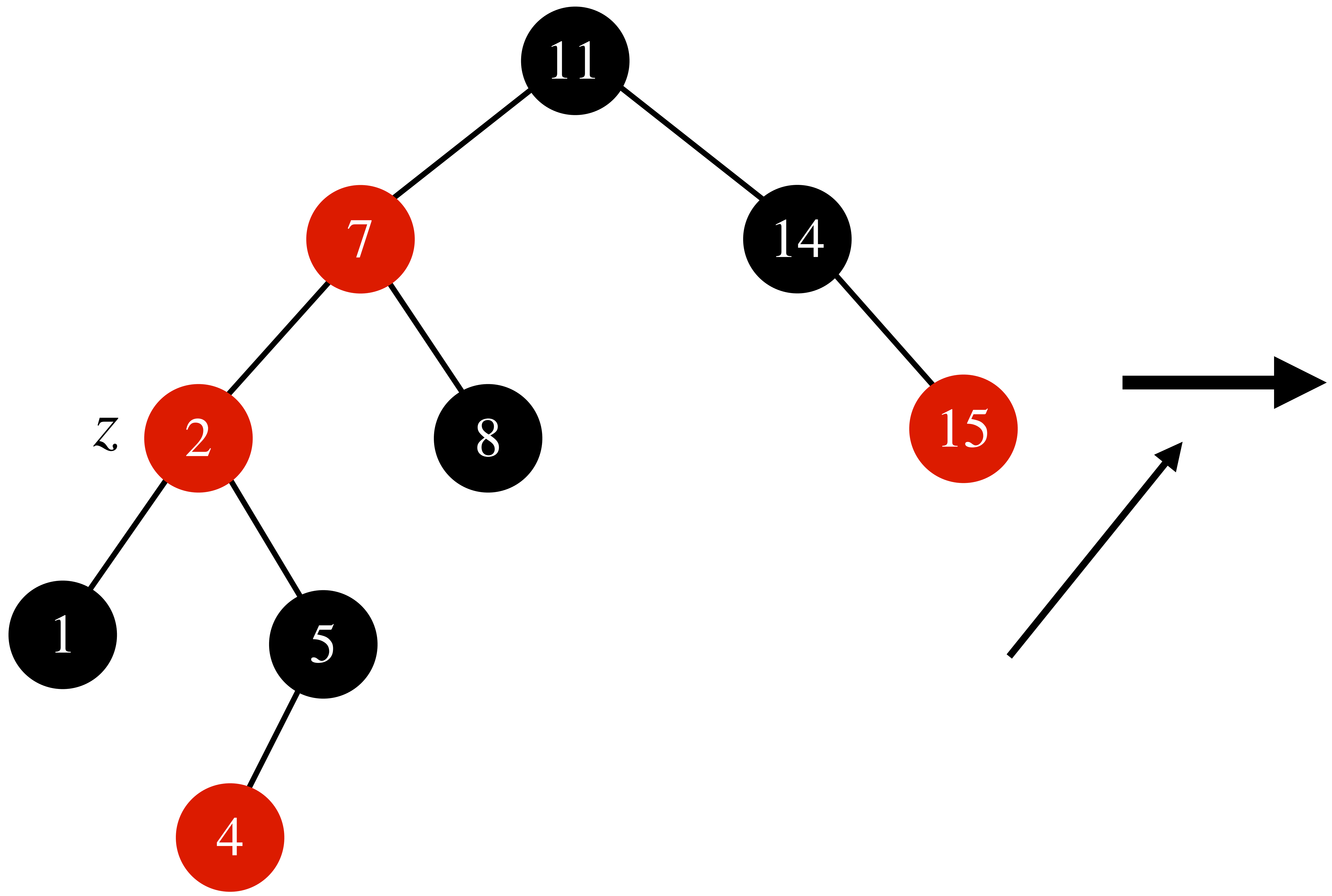
RB-Trees: Insertion Case 3

Case 3**:** z's uncle is black and z is a left child.



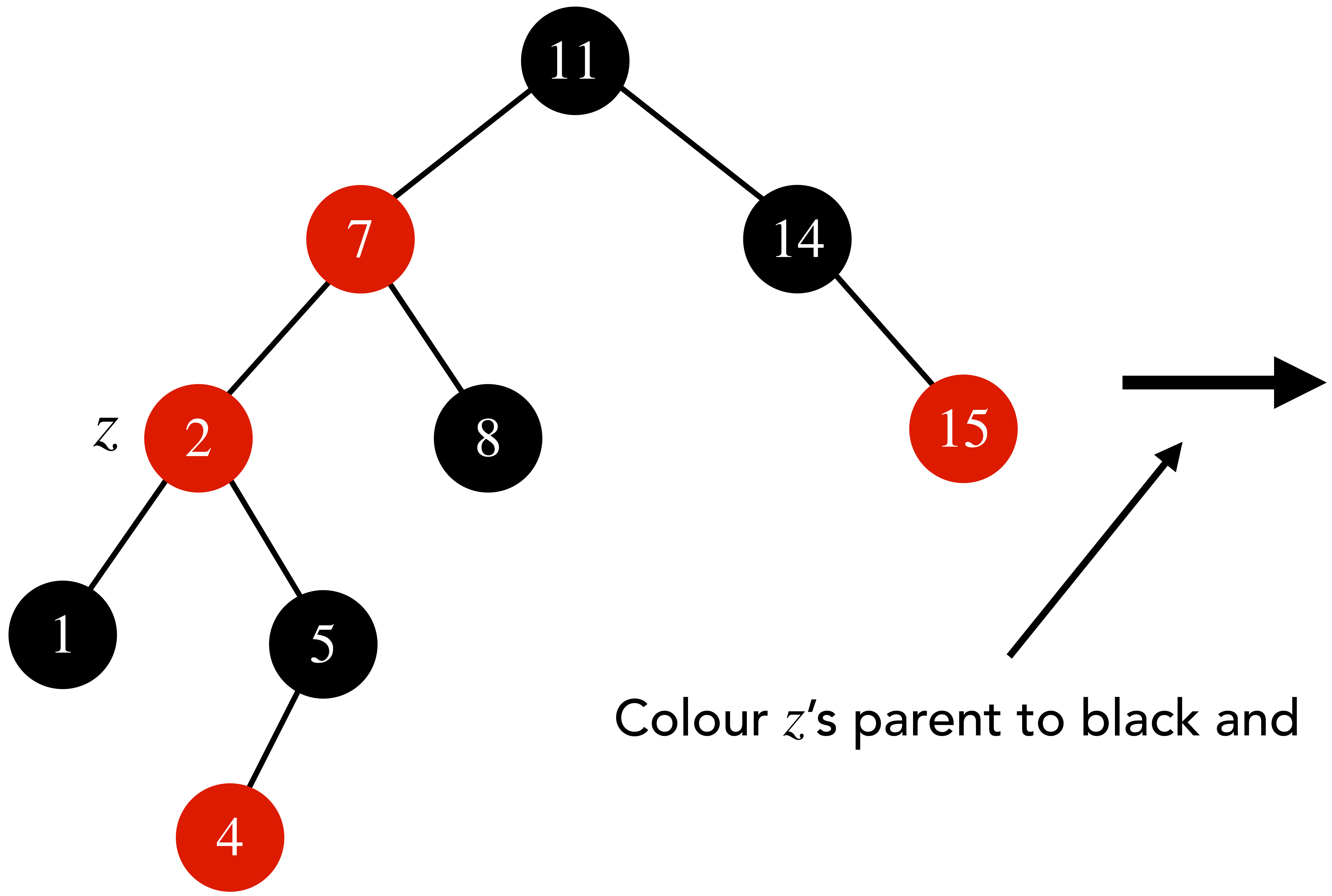
RB-Trees: Insertion Case 3

Case 3**:** z's uncle is black and z is a **left child**.



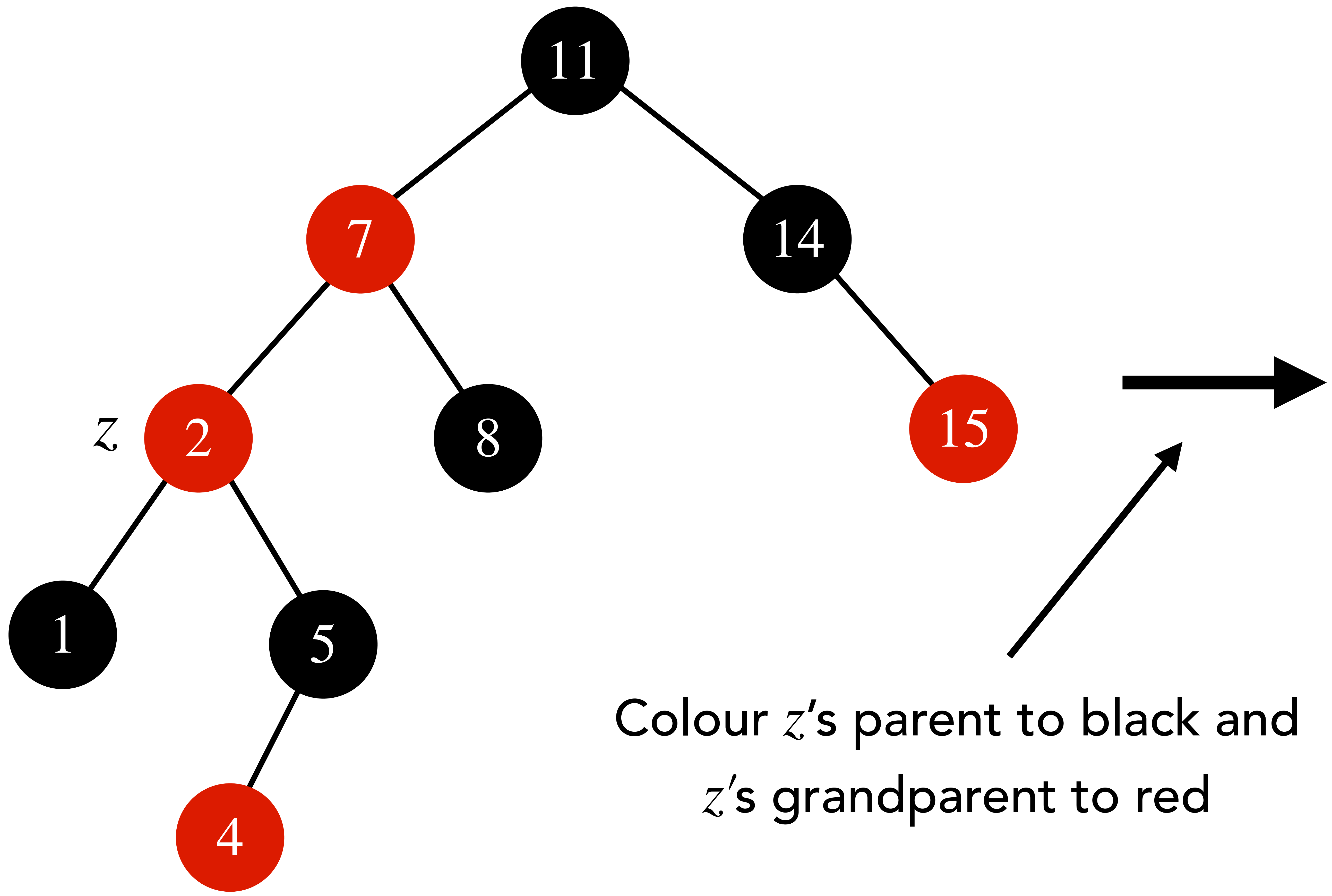
RB-Trees: Insertion Case 3

Case 3**:** z's uncle is black and z is a left child.



RB-Trees: Insertion Case 3

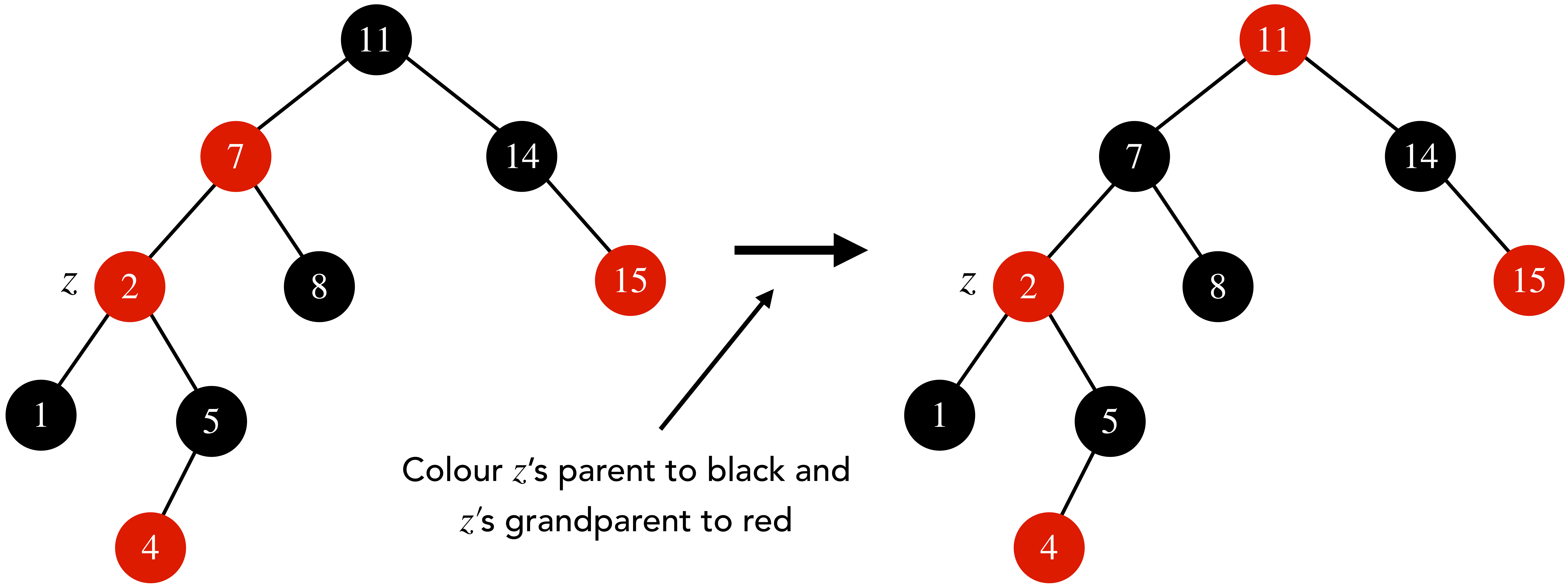
Case 3**:** z's uncle is **black** and z is a **left child**.



Colour z's parent to black and z's grandparent to red

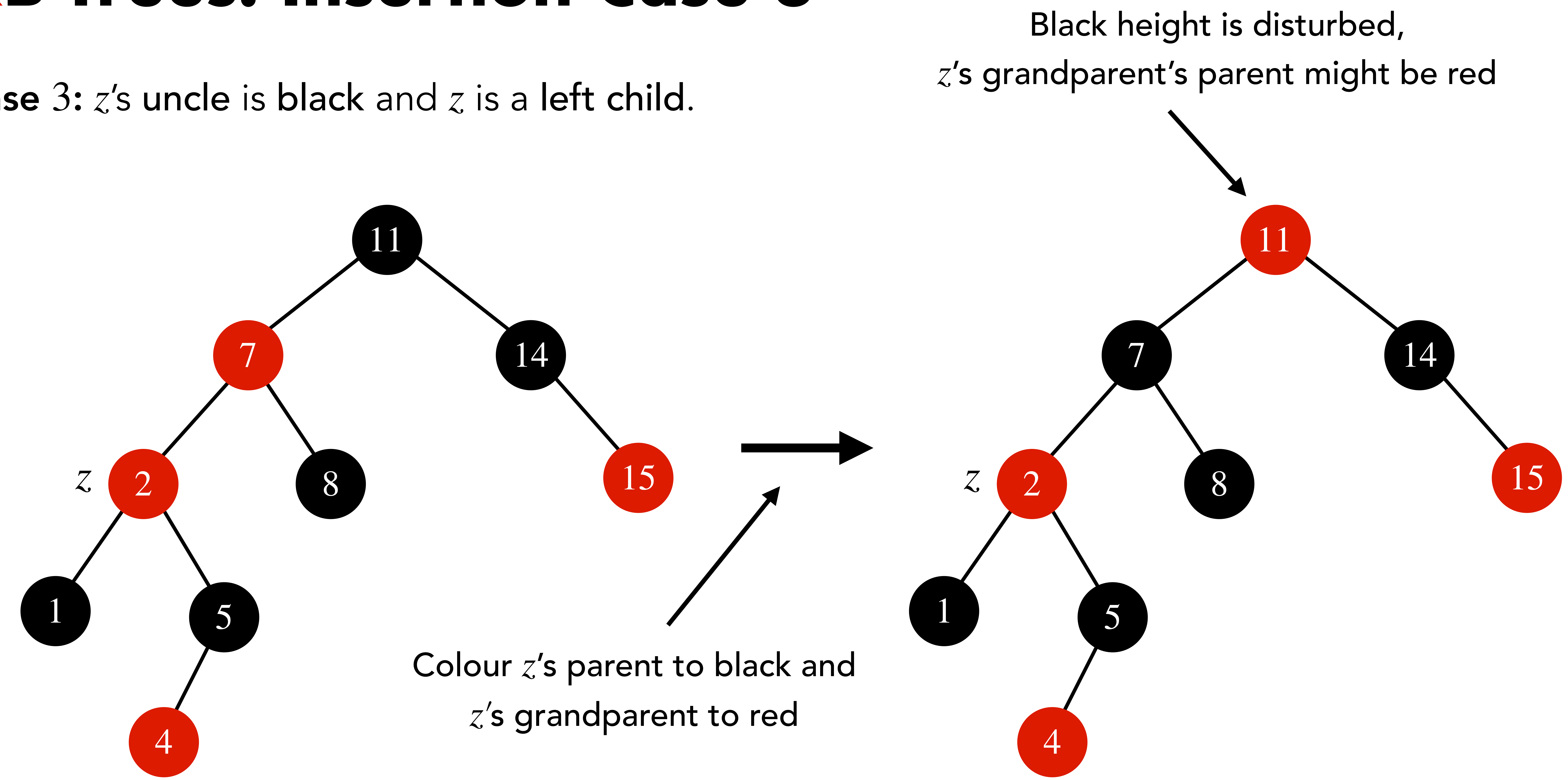
RB-Trees: Insertion Case 3

Case 3**:** z's uncle is **black** and z is a **left child**.



RB-Trees: Insertion Case 3

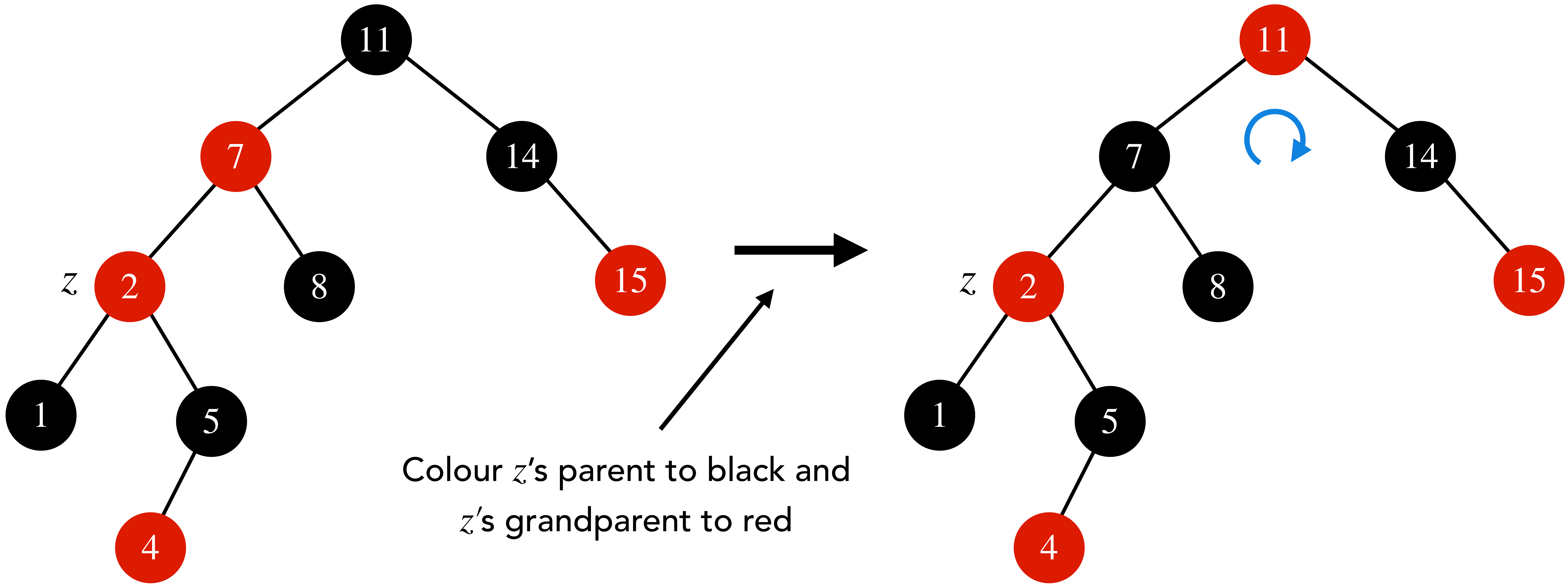
Case 3**:** z's uncle is black and z is a **left child**.



RB-Trees: Insertion Case 3

Case 3**:** z's uncle is black and z is a **left child**.

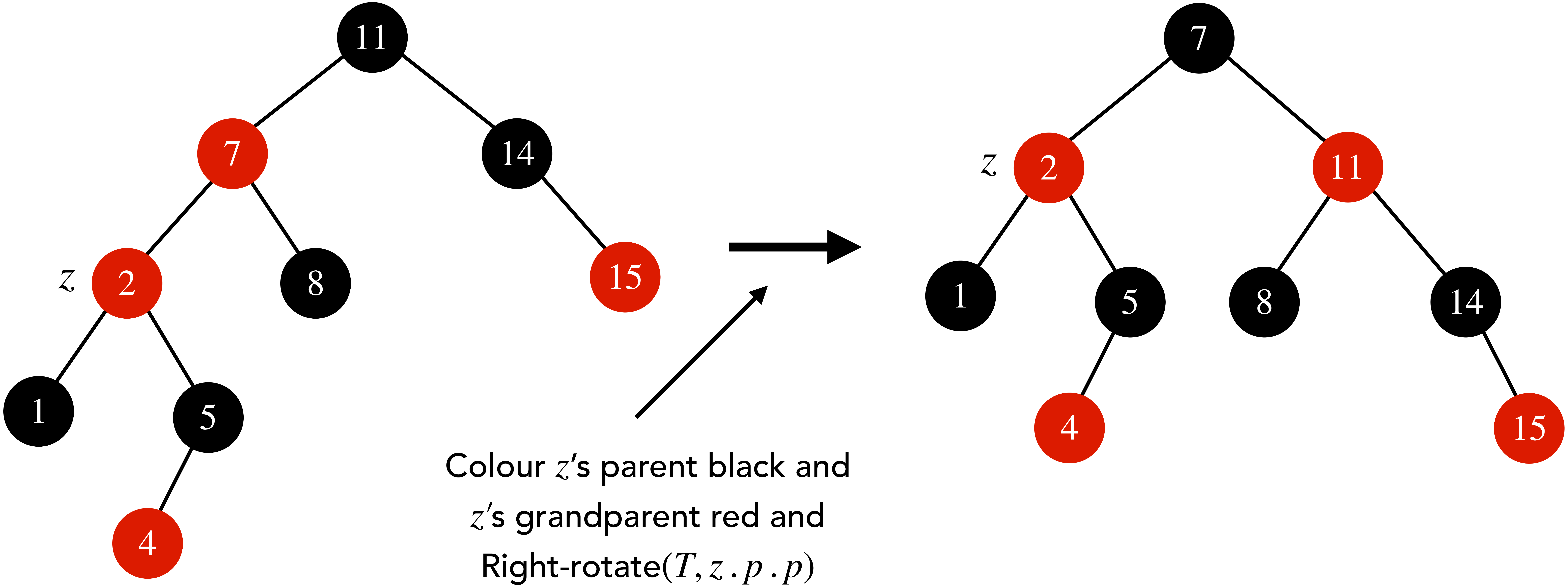
Black height is disturbed,
z's grandparent's parent might be red



Colour z's parent to black and
z's grandparent to red

RB-Trees: Insertion Case 3

Case 3: z's uncle is black and z is a **left child**.



Colour z's parent black and z's grandparent red and Right-rotate$(T, z.p.p)$

RB-Trees: Deletion

RB-Trees: Deletion

Two stages of deletion:

RB-Trees: Deletion

Two stages of deletion:

- Delete the node as we do in a BST.

RB-Trees: Deletion

Two stages of deletion:

- Delete the node as we do in a BST.
- Do fix-ups as deletion may cause a violation of a few Red-blue properties.

RB-Trees: Deletion

Two stages of deletion:

- Delete the node as we do in a BST.
- Do fix-ups as deletion may cause a violation of a few Red-blue properties.

Let's recall deletion in a BST and spot special nodes, y and x.

RB-Trees: Deletion

Two stages of deletion:

- Delete the node as we do in a BST.
- Do fix-ups as deletion may cause a violation of a few Red-blue properties.

Let's recall deletion in a BST and spot special nodes, y and x.

y will be the node we will "actually" be taking out
and whether fix ups are require will depend on the colour of y

RB-Trees: Deletion

Two stages of deletion:

- Delete the node as we do in a BST.
- Do fix-ups as deletion may cause a violation of a few Red-blue properties.

Let's recall deletion in a BST and spot special nodes, y and x.

Fix ups will start from x after removing y

Recall Deletion in BSTs

Recall Deletion in BSTs

Let z be the node we want to delete.

Recall Deletion in BSTs

Let z be the node we want to delete. Then, the following cases are possible:

Recall Deletion in BSTs

Let z be the node we want to delete. Then, the following cases are possible:

- **Case** 1: z has no children.

Recall Deletion in BSTs

Let z be the node we want to delete. Then, the following cases are possible:

- **Case** 1: z has no children.
- **Case** 2: z has only single child.

Recall Deletion in BSTs

Let z be the node we want to delete. Then, the following cases are possible:

- **Case** 1: z has no children.
- **Case** 2: z has only single child.
- **Case** 3: z has two children.

Recall Deletion in BSTs

Let z be the node we want to delete. Then, the following cases are possible:

- **Case** 1: z has no children.
- **Case** 2: z has only single child.
- **Case** 3: z has two children.

Easy

Recall Deletion in BSTs

Let z be the node we want to delete. Then, the following cases are possible:

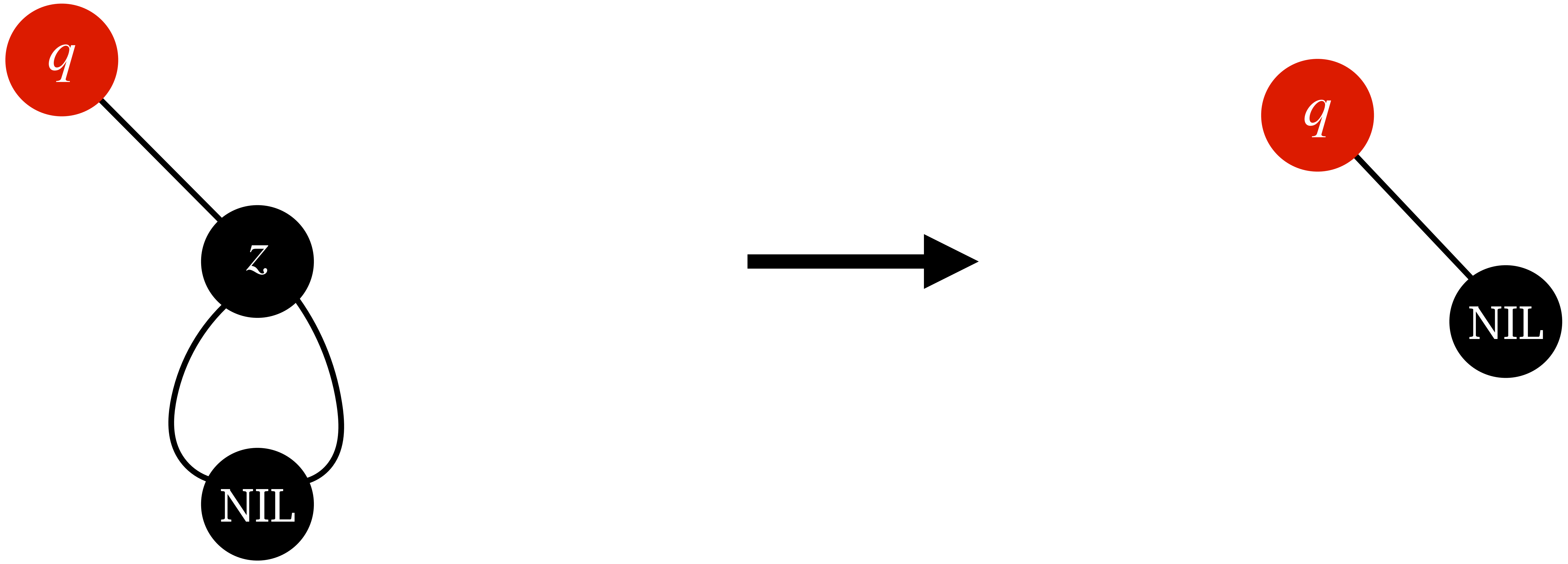
- **Case** 1: z has no children.
- **Case** 2: z has only single child.
- **Case** 3: z has two children.

Easy

Not so easy

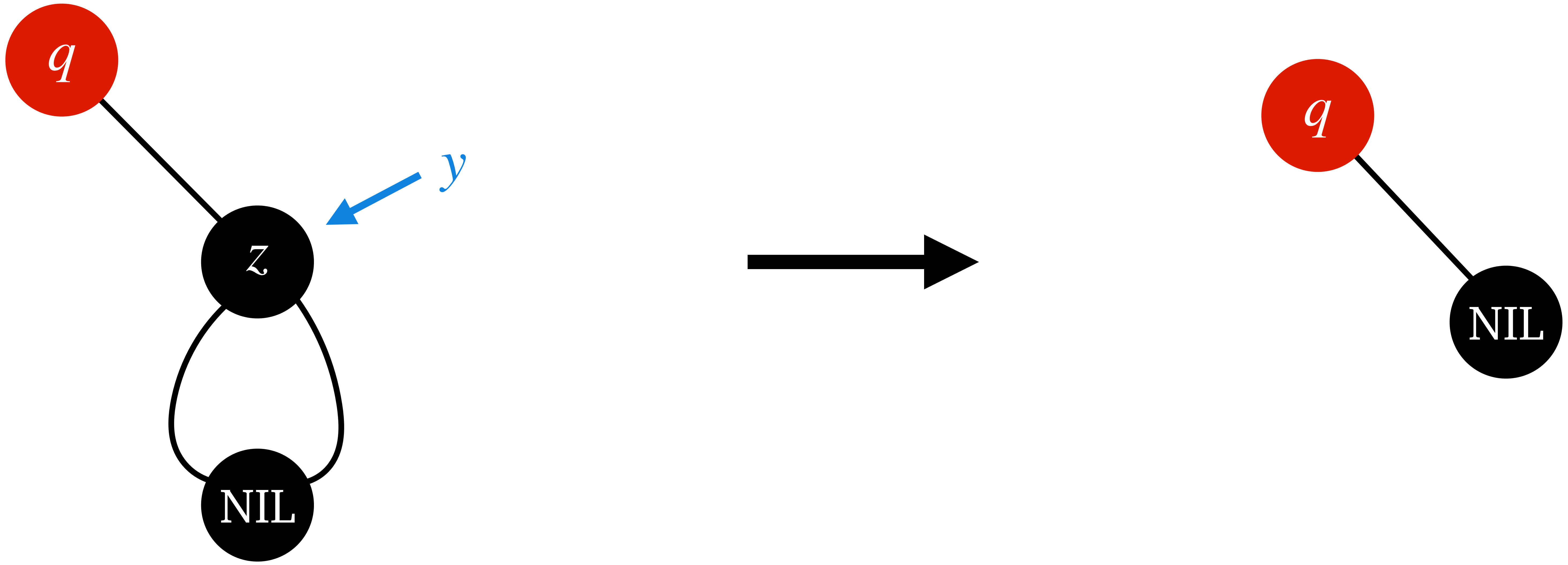
RB-Trees: Deletion

Case 1: z has no (non-NIL) children. (WLOG assume z is a right child.)



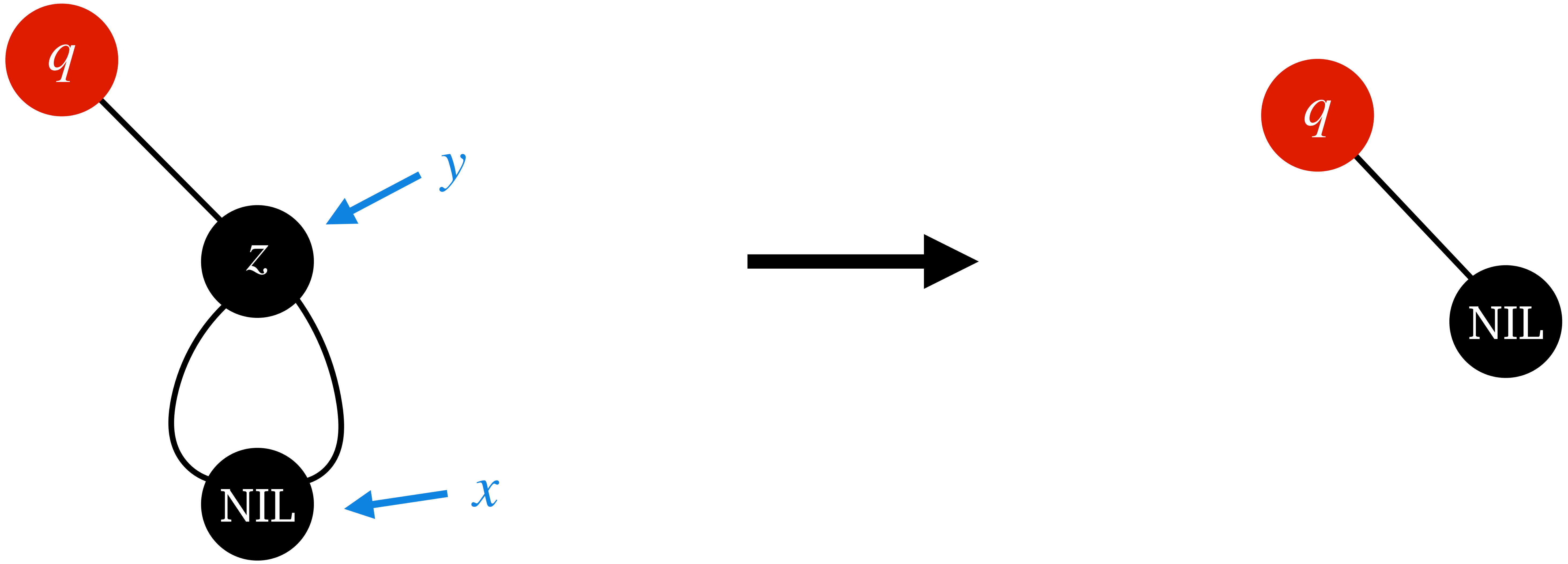
RB-Trees: Deletion

Case 1: z has no (non-NIL) children. (WLOG assume z is a right child.)



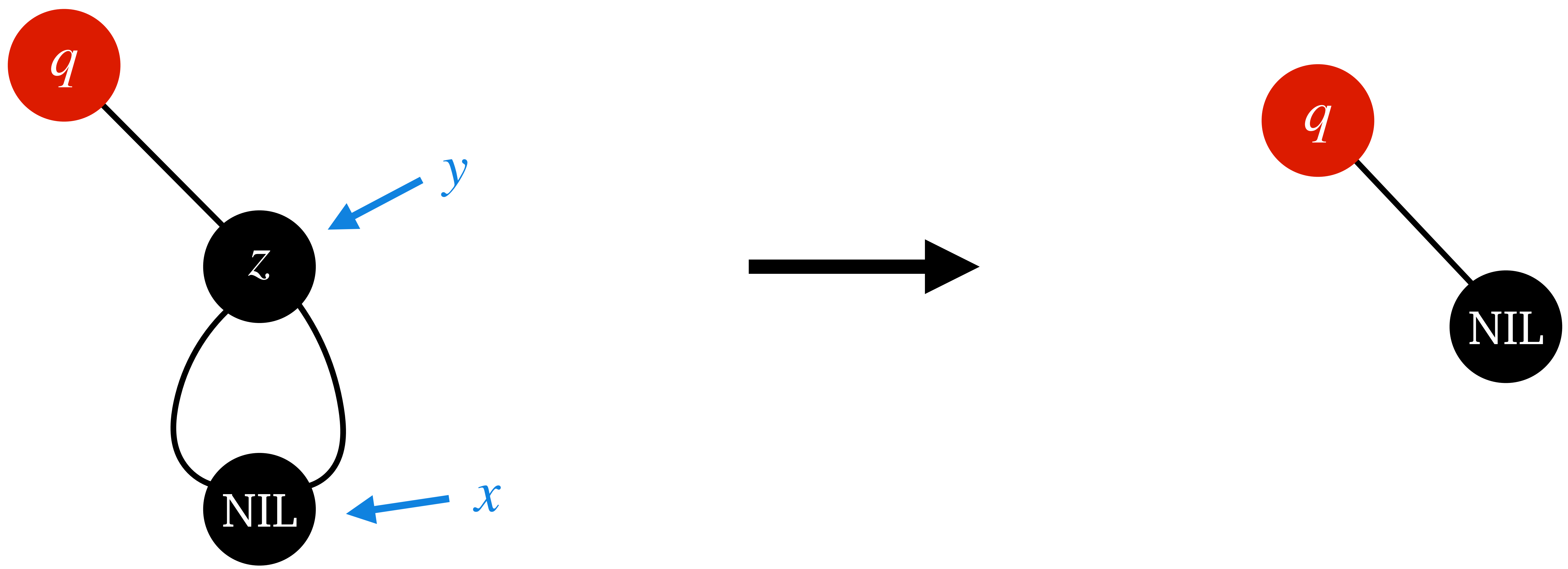
RB-Trees: Deletion

Case 1: z has no (non-NIL) children. (WLOG assume z is a right child.)



RB-Trees: Deletion

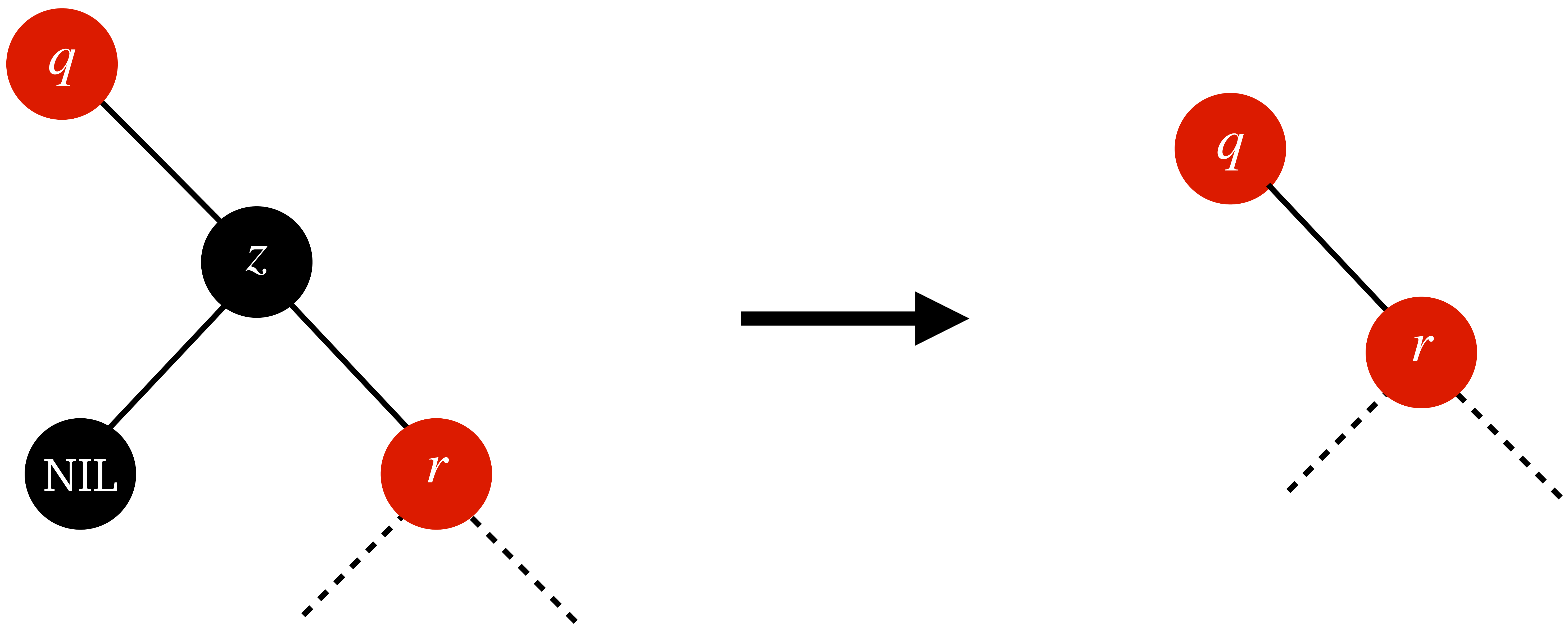
Case 1: z has no (non-NIL) children. (WLOG assume z is a right child.)



Note: In this case, y is z and x is NIL.

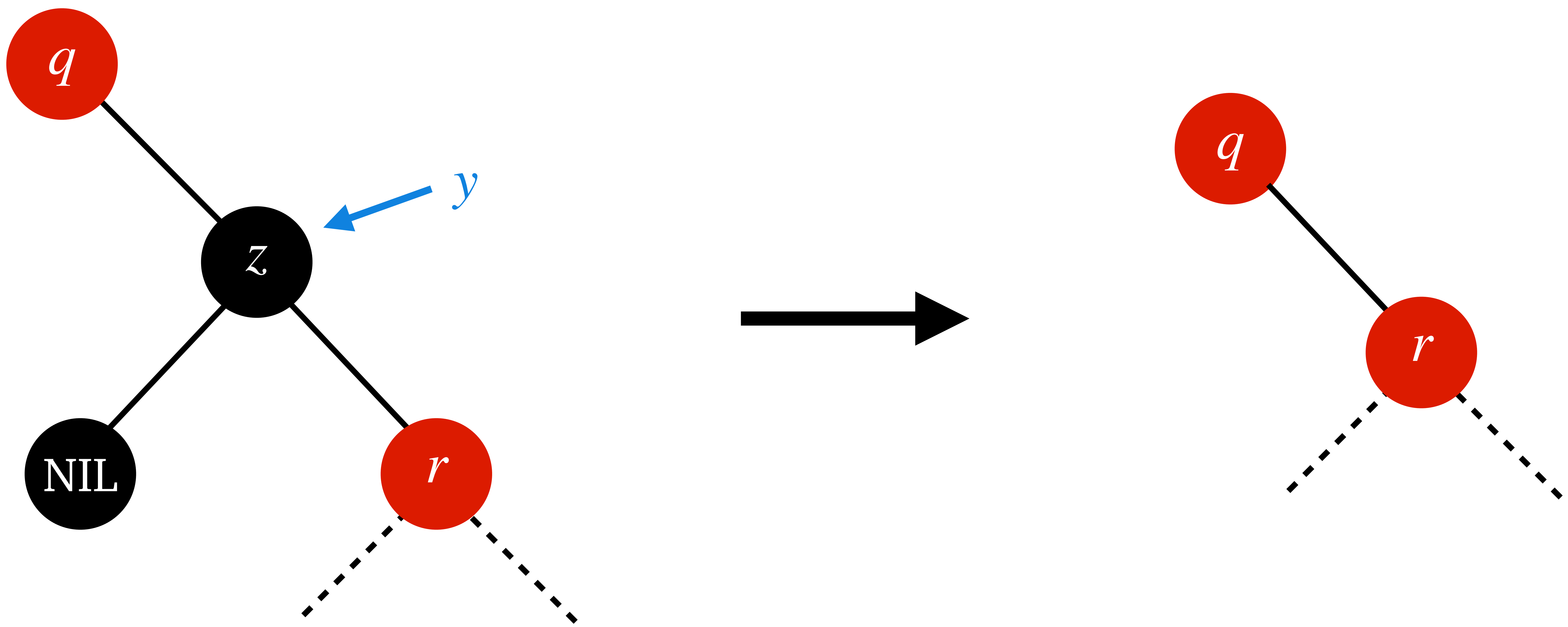
RB-Trees: Deletion

Case 2: z has one (non-NIL) child. (WLOG assume z is a right child.)



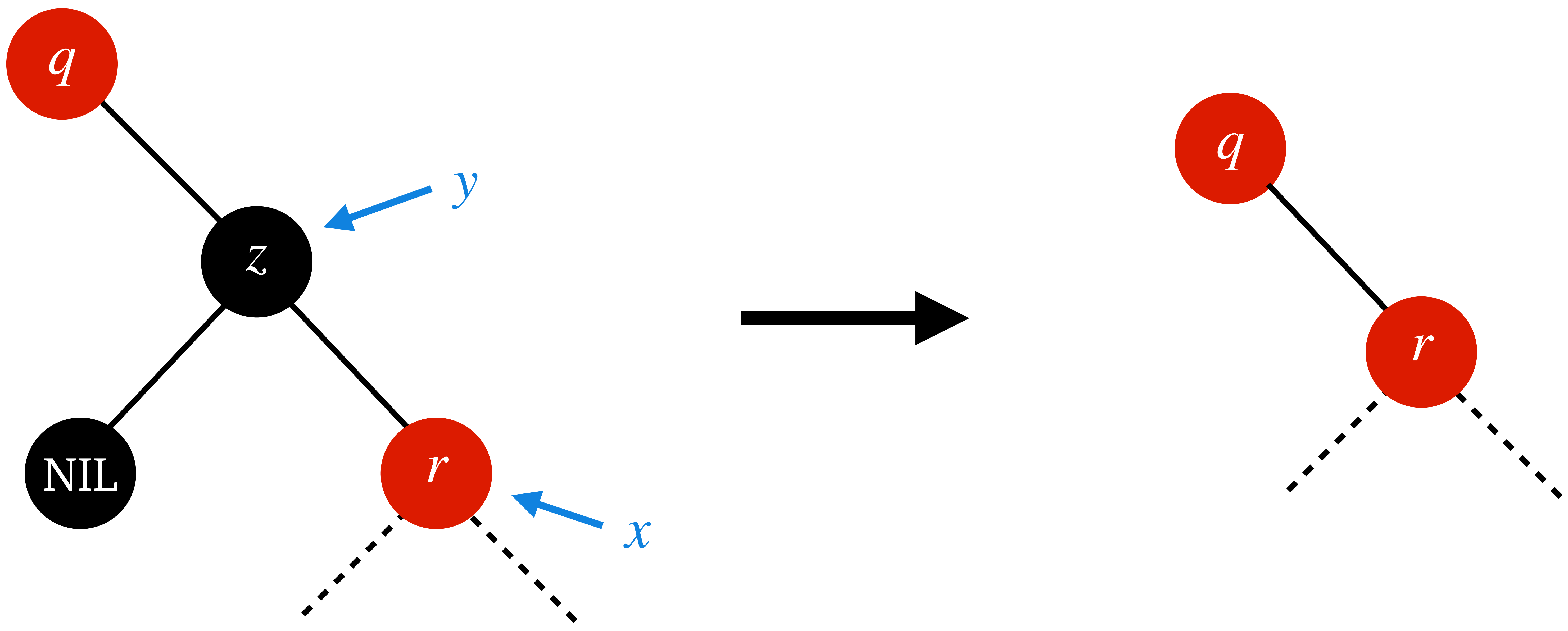
RB-Trees: Deletion

Case 2: z has one (non-NIL) child. (WLOG assume z is a right child.)



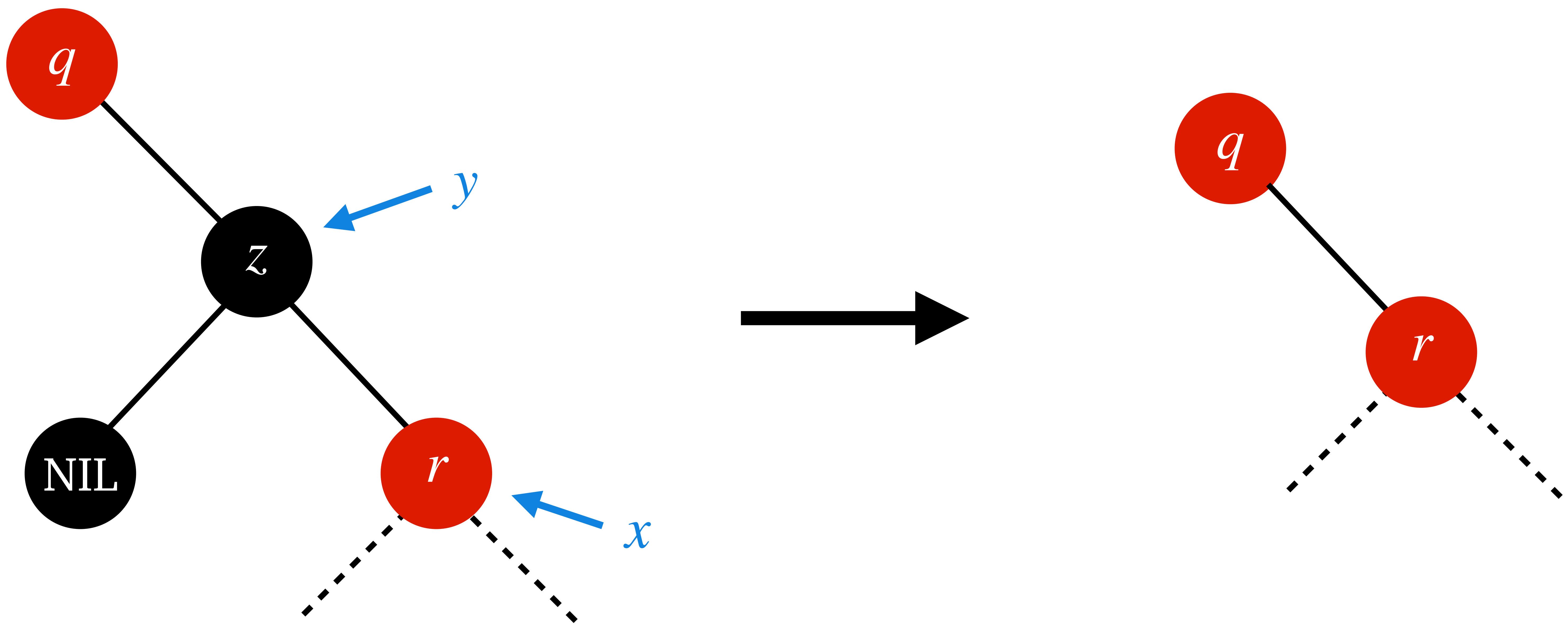
RB-Trees: Deletion

Case 2: z has one (non-NIL) child. (WLOG assume z is a right child.)



RB-Trees: Deletion

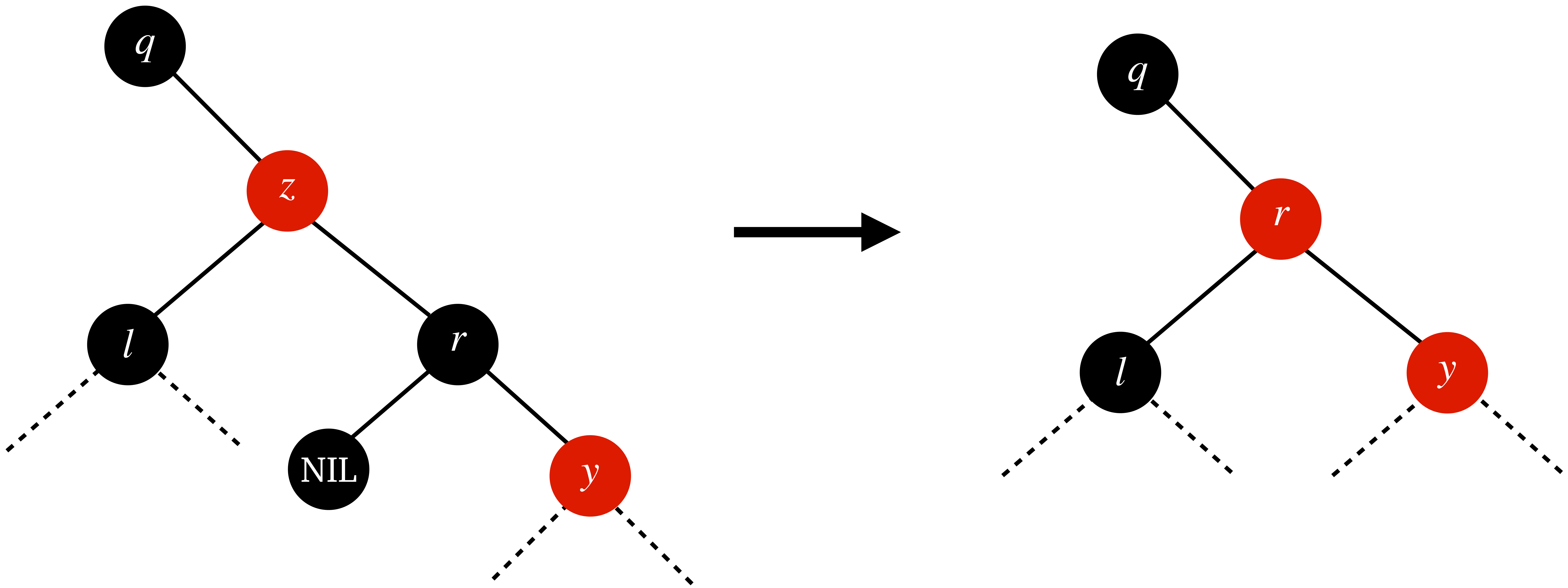
Case 2: z has one (non-NIL) child. (WLOG assume z is a right child.)



Note: In this case, y is z and x is the only child of z.

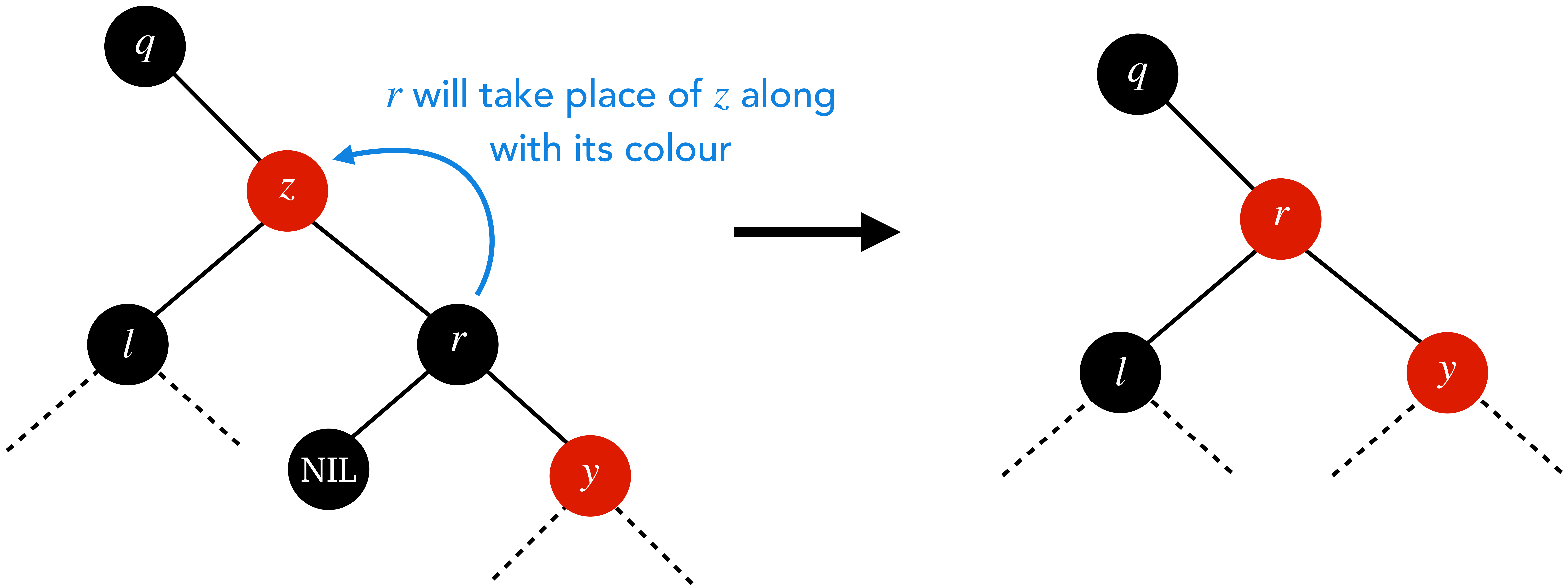
RB-Trees: Deletion

Case 3**a**: z has two (non-NIL) children where its right child has no left child.



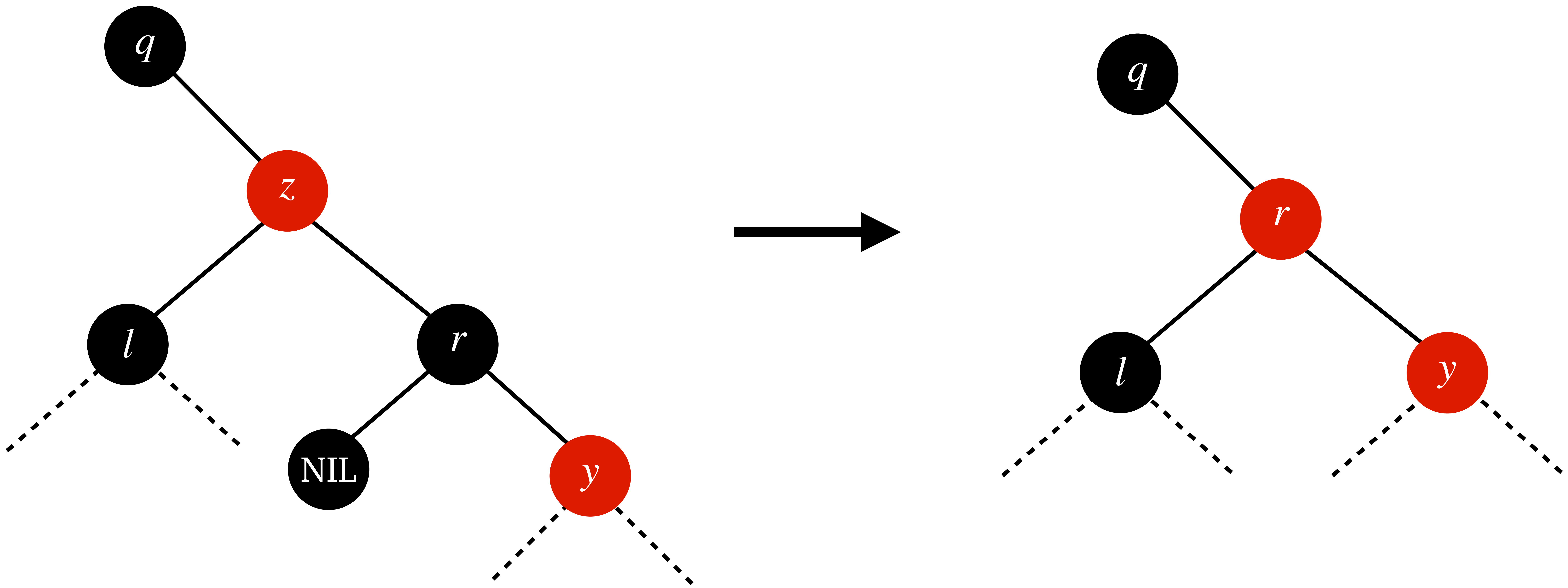
RB-Trees: Deletion

Case 3**a**: z has two (non-NIL) children where its right child has no left child.



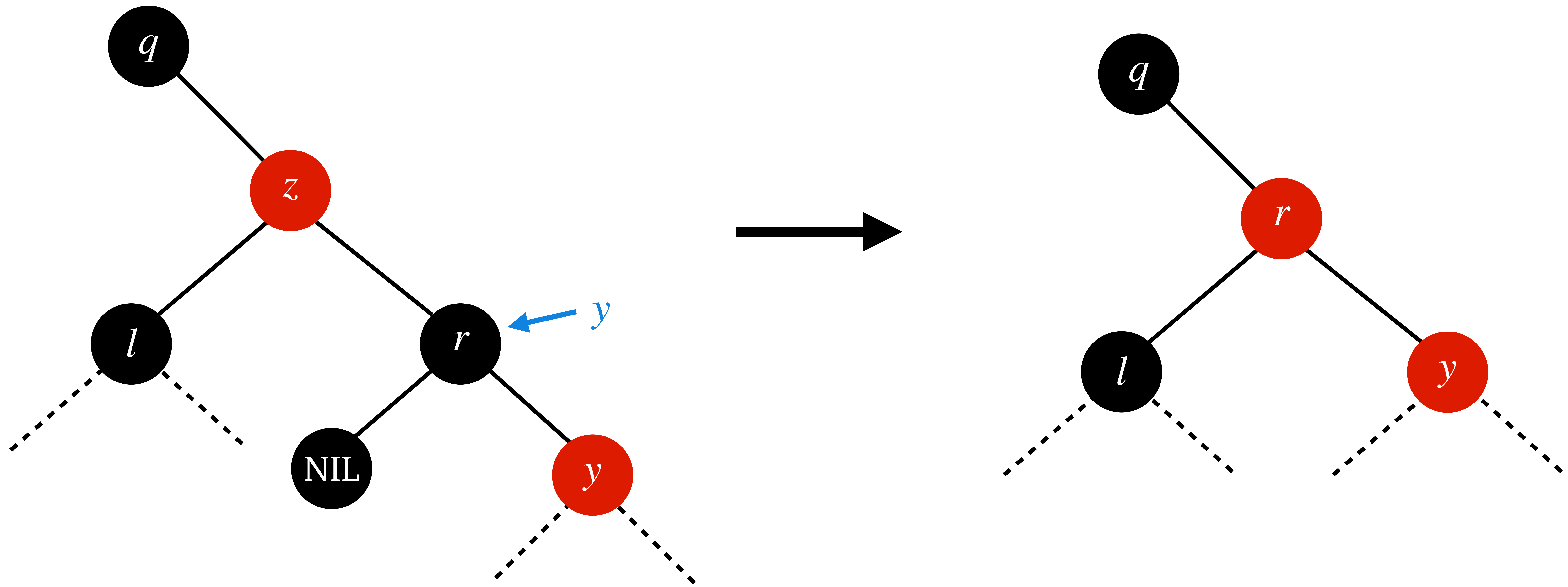
RB-Trees: Deletion

Case 3**a**: z has two (non-NIL) children where its right child has no left child.



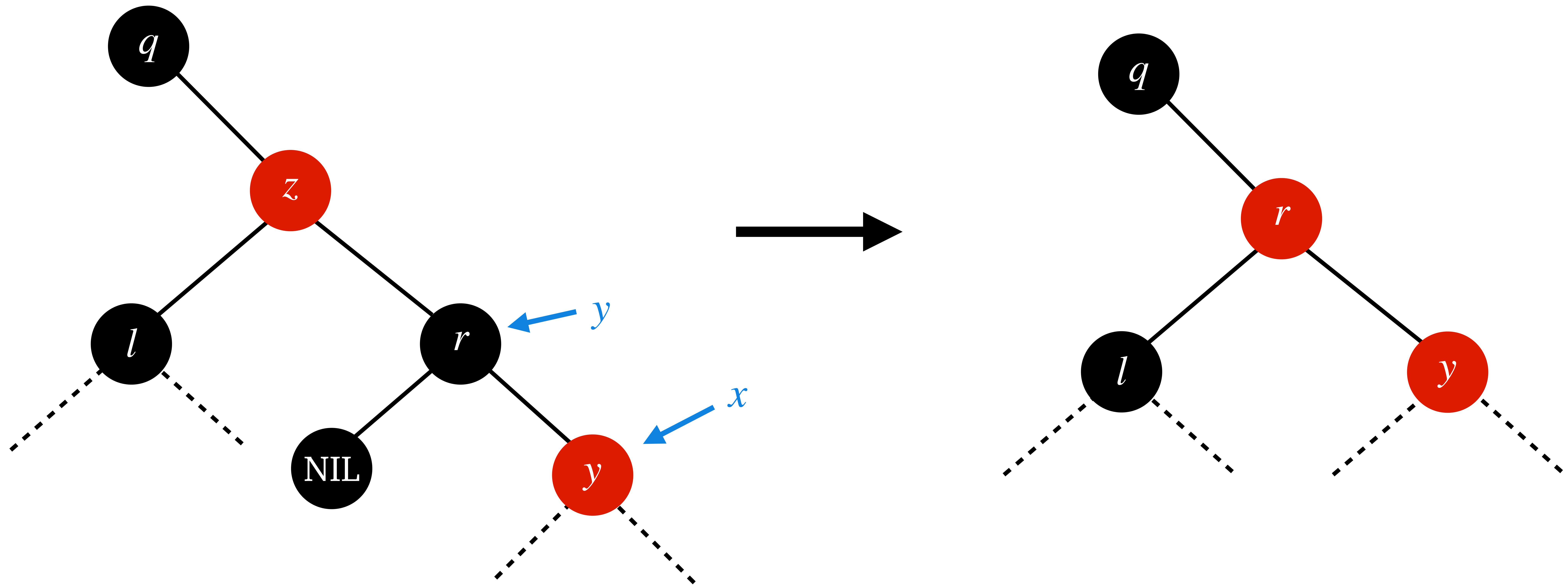
RB-Trees: Deletion

Case 3**a**: z has two (non-NIL) children where its right child has no left child.



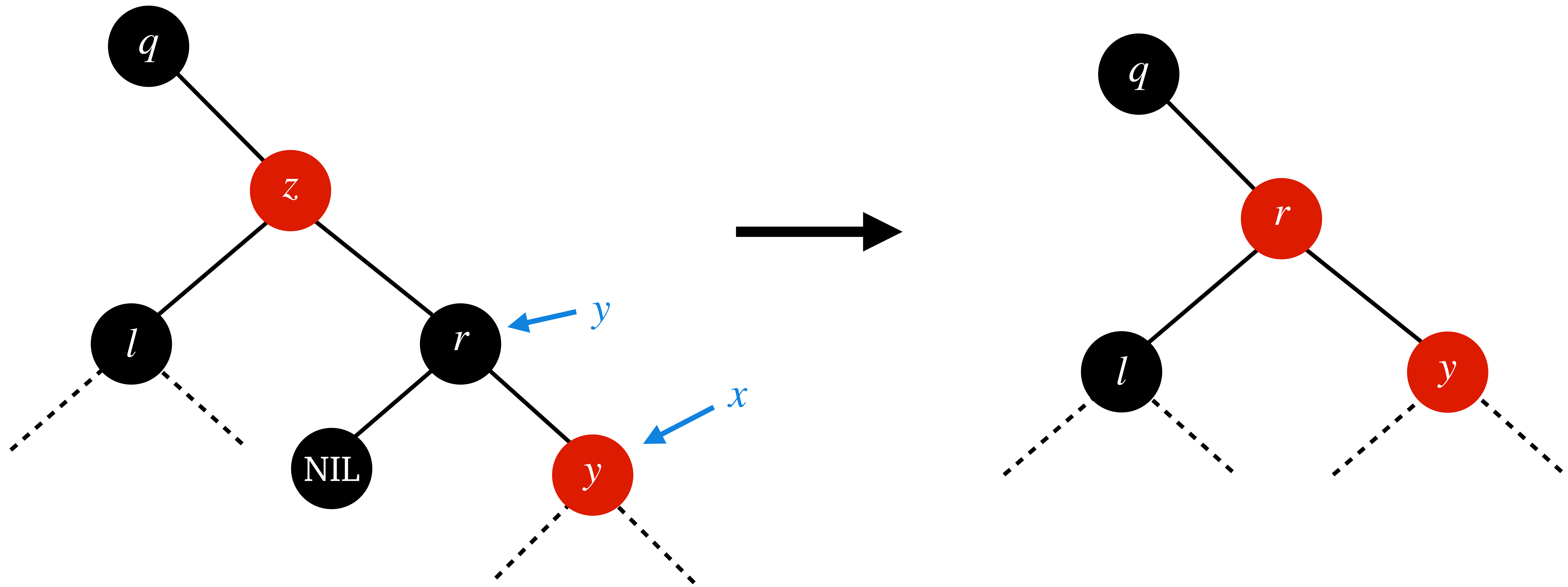
RB-Trees: Deletion

Case 3**a**: z has two (non-NIL) children where its right child has no left child.



RB-Trees: Deletion

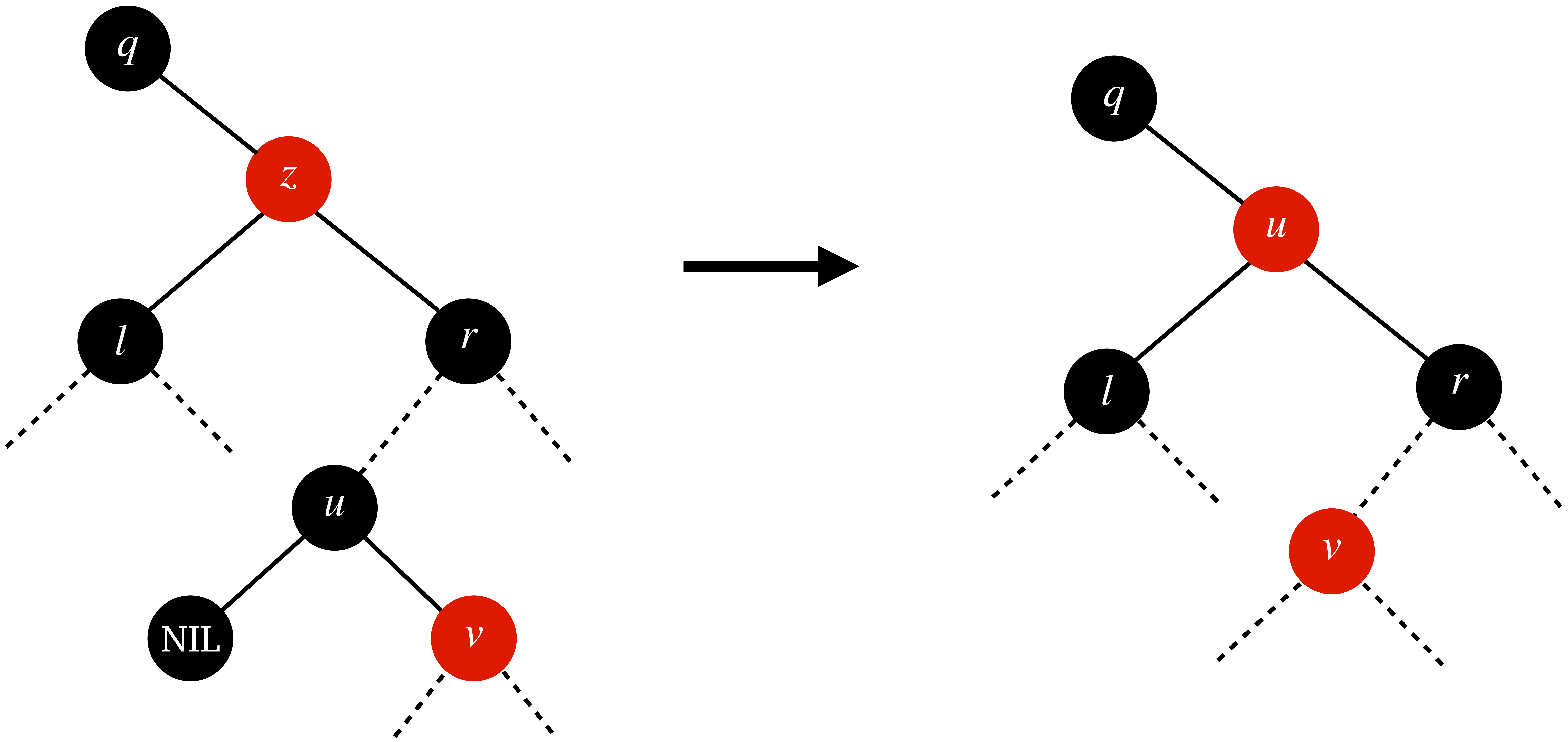
Case 3a: z has two (non-NIL) children where its right child has no left child.



Note: In this case, y is the successor of z and x is either NIL or the only child of y.

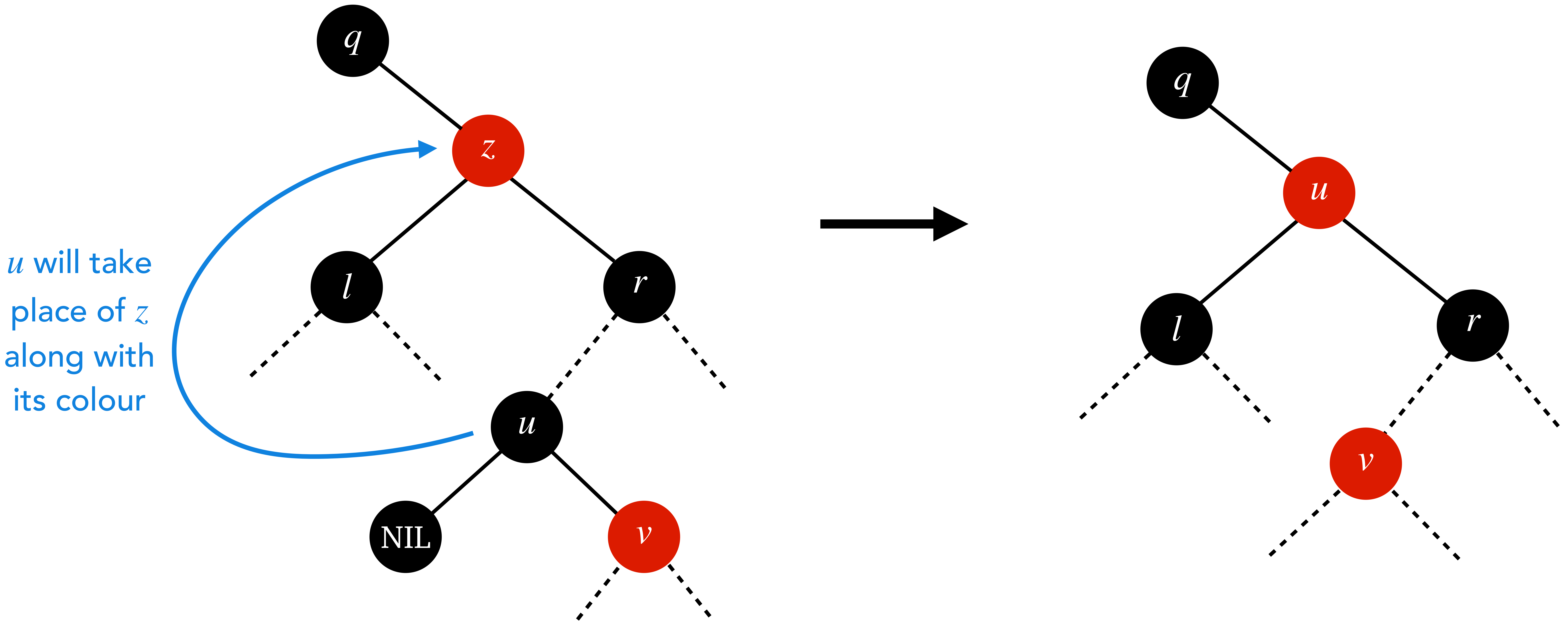
RB-Trees: Deletion

Case 3**b**: z has two (non-NIL) children where its right child has a left child.



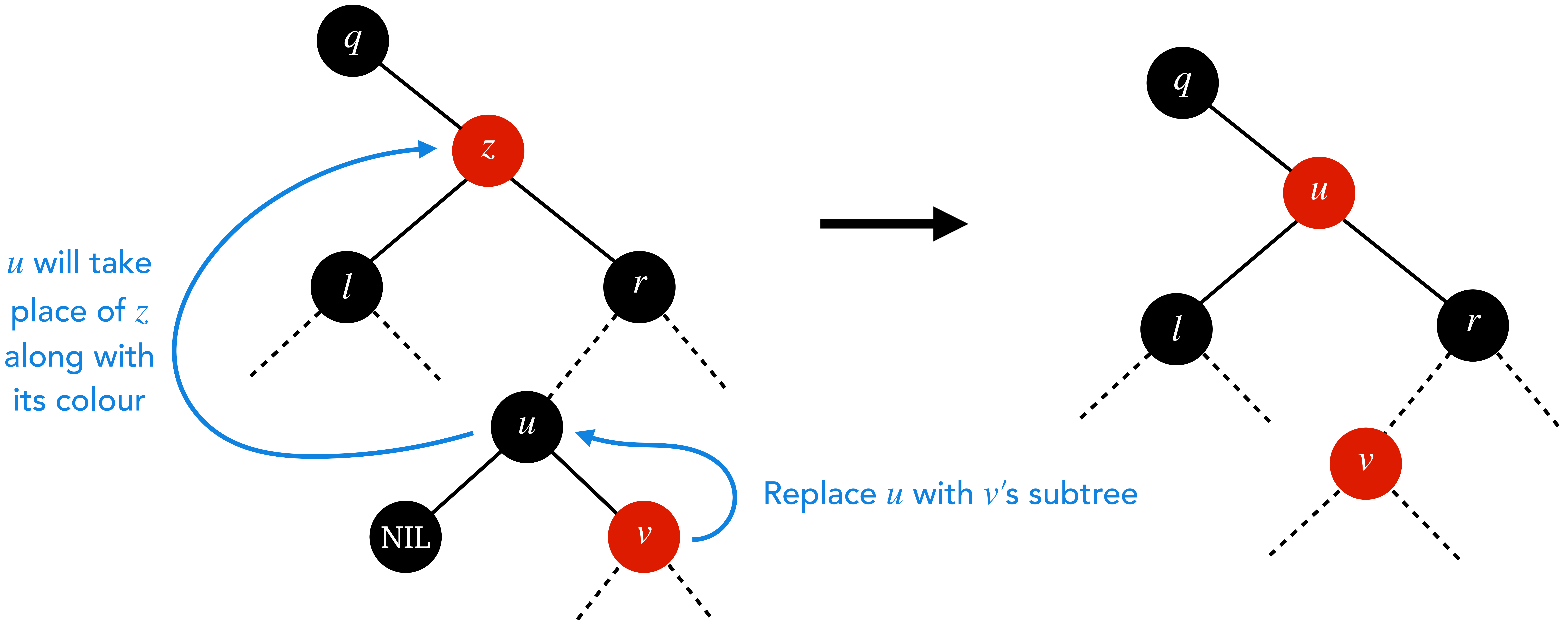
RB-Trees: Deletion

Case 3**b**: z has two (non-NIL) children where its right child has a left child.



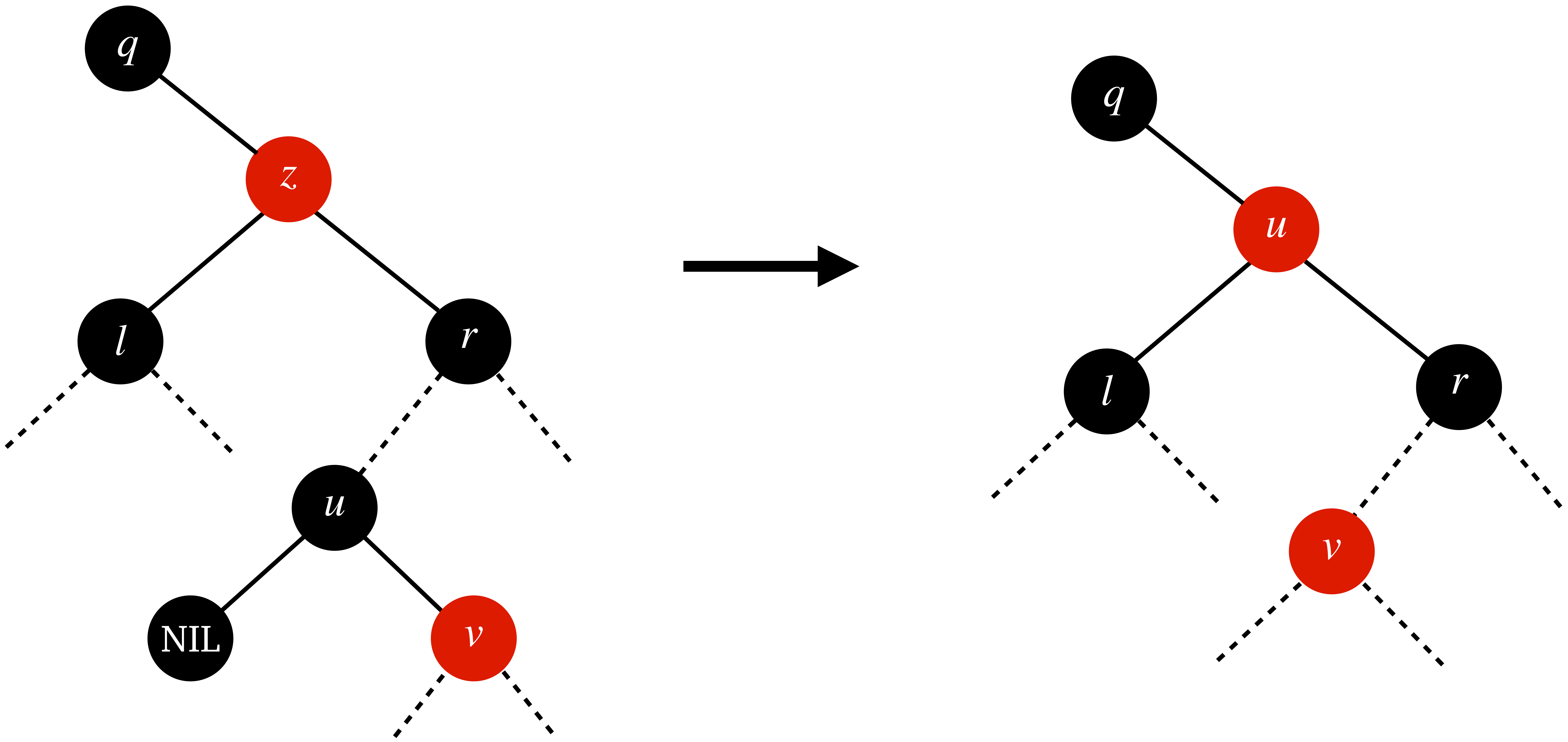
RB-Trees: Deletion

Case 3**b**: z has two (non-NIL) children where its right child has a left child.



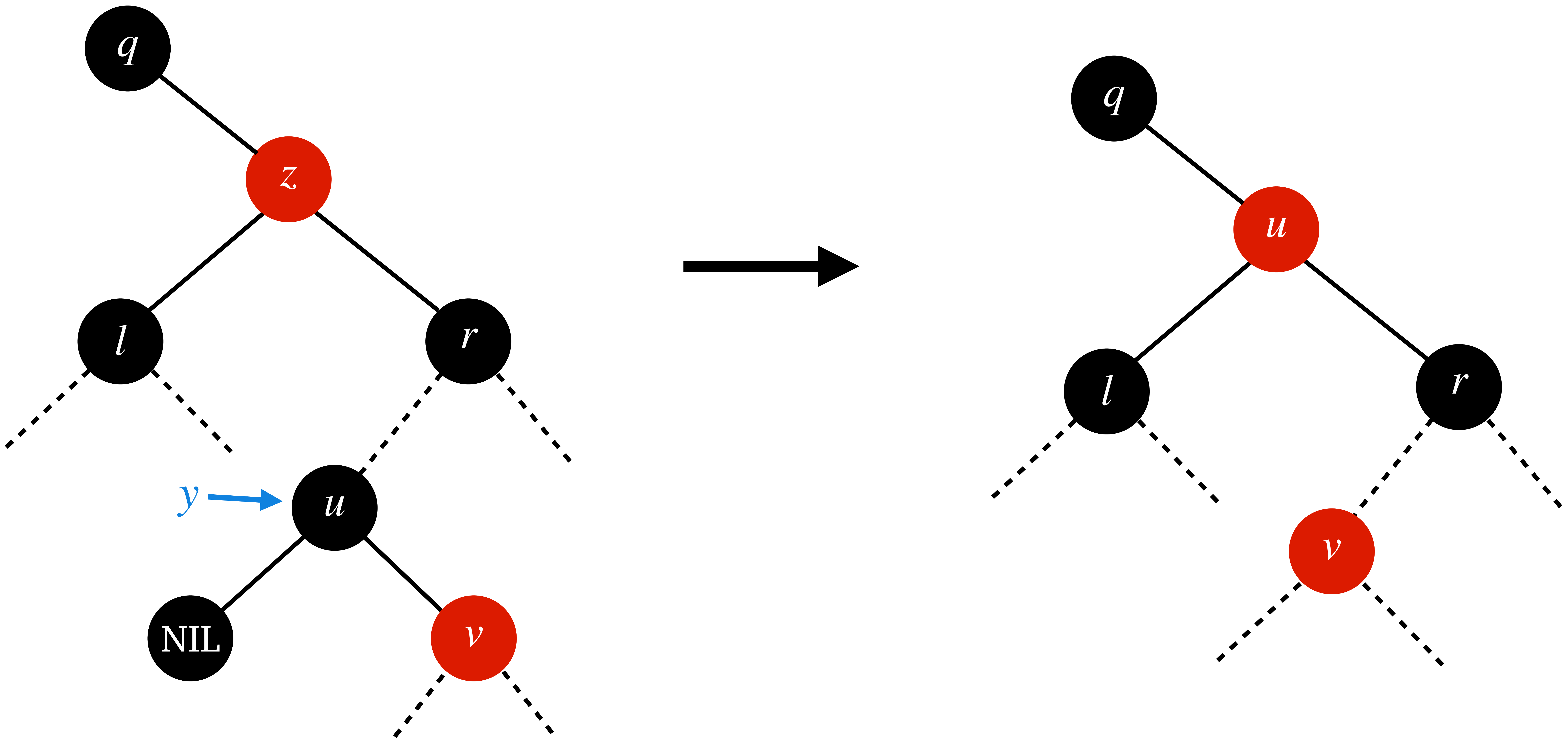
RB-Trees: Deletion

Case 3**b:** z has two (non-NIL) children where its right child has a left child.



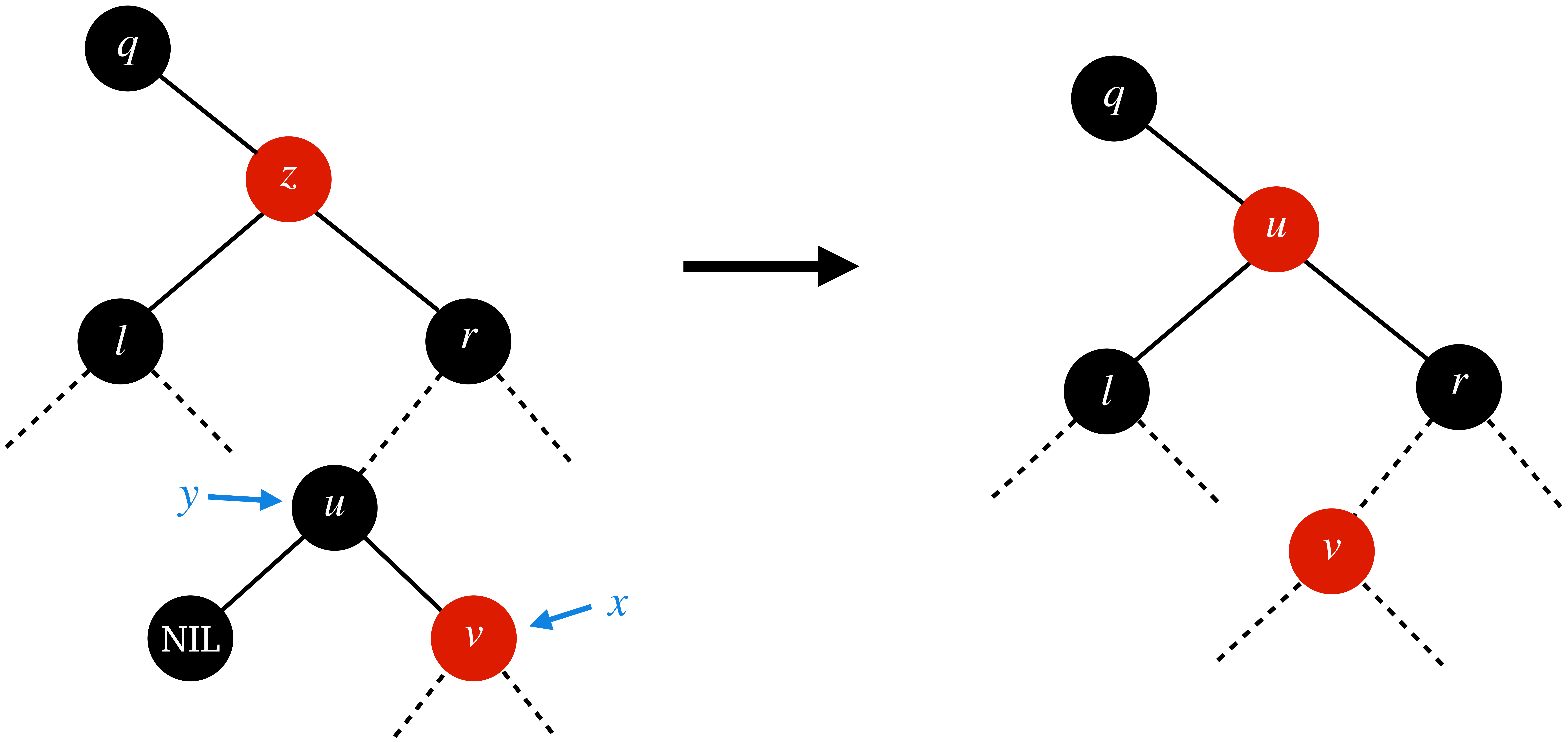
RB-Trees: Deletion

Case 3b: z has two (non-NIL) children where its right child has a left child.



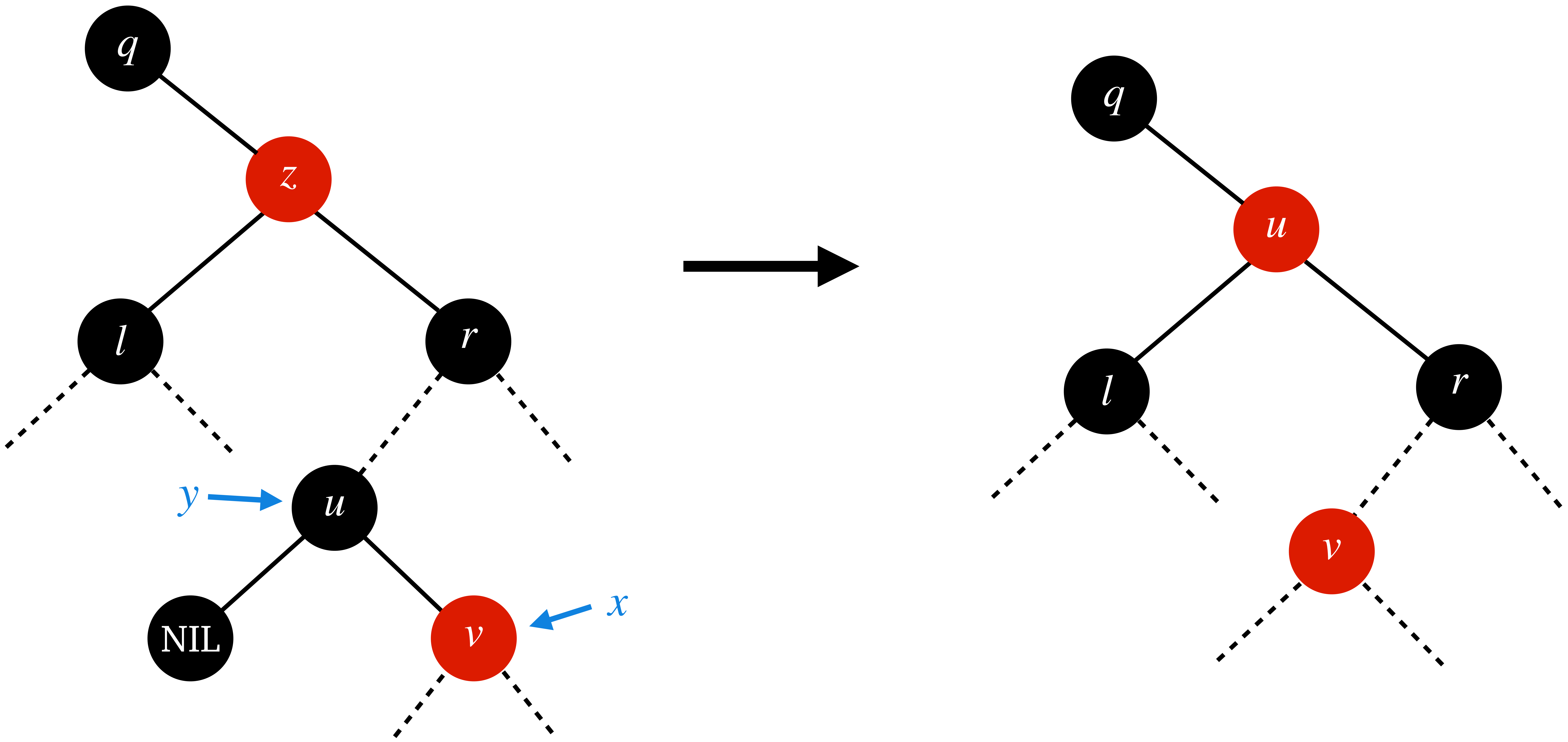
RB-Trees: Deletion

Case 3**b**: z has two (non-NIL) children where its right child has a left child.



RB-Trees: Deletion

Case 3b: z has two (non-NIL) children where its right child has a left child.



Note: In this case, y is the successor of z and x is either NIL or the only child of y.

RB-Trees: Deletion

RB-Trees: Deletion

Let z be the node we want to delete:

RB-Trees: Deletion

Let z be the node we want to delete:

- **Case 1:** If z has no (non-NIL) child, then $y = z$ and x will be NIL.

RB-Trees: Deletion

Let z be the node we want to delete:

- **Case 1:** If z has no (non-NIL) child, then $y = z$ and x will be NIL.
- **Case 2:** If z has exactly one (non-NIL) child, then $y = z$ and x will be y's only non-NIL child.

RB-Trees: Deletion

Let z be the node we want to delete:

- **Case 1:** If z has no (non-NIL) child, then $y = z$ and x will be NIL.
- **Case 2:** If z has exactly one (non-NIL) child, then $y = z$ and x will be y's only non-NIL child.
- **Case 3:** Else, y will be the successor of z and x will be y's only non-NIL child or NIL.

RB-Trees: Deletion

Let z be the node we want to delete:

- **Case 1:** If z has no (non-NIL) child, then $y = z$ and x will be NIL.
- **Case 2:** If z has exactly one (non-NIL) child, then $y = z$ and x will be y's only non-NIL child.
- **Case 3:** Else, y will be the successor of z and x will be y's only non-NIL child or NIL.

Skeleton for Deletion:

RB-Trees: Deletion

Let z be the node we want to delete:

- **Case 1:** If z has no (non-NIL) child, then $y = z$ and x will be NIL.
- **Case 2:** If z has exactly one (non-NIL) child, then $y = z$ and x will be y's only non-NIL child.
- **Case 3:** Else, y will be the successor of z and x will be y's only non-NIL child or NIL.

Skeleton for Deletion:

- Find y and x.

RB-Trees: Deletion

Let z be the node we want to delete:

- **Case 1:** If z has no (non-NIL) child, then $y = z$ and x will be NIL.
- **Case 2:** If z has exactly one (non-NIL) child, then $y = z$ and x will be y's only non-NIL child.
- **Case 3:** Else, y will be the successor of z and x will be y's only non-NIL child or NIL.

Skeleton for Deletion:

- Find y and x.
- If it's **Case 3**, replace z with y.

RB-Trees: Deletion

Let z be the node we want to delete:

- **Case 1:** If z has no (non-NIL) child, then $y = z$ and x will be NIL.
- **Case 2:** If z has exactly one (non-NIL) child, then $y = z$ and x will be y's only non-NIL child.
- **Case 3:** Else, y will be the successor of z and x will be y's only non-NIL child or NIL.

Skeleton for Deletion:

- Find y and x.
- If it's **Case 3**, replace z with y.
- Remove y.

RB-Trees: Deletion

Let z be the node we want to delete:

- **Case 1:** If z has no (non-NIL) child, then $y = z$ and x will be NIL.
- **Case 2:** If z has exactly one (non-NIL) child, then $y = z$ and x will be y's only non-NIL child.
- **Case 3:** Else, y will be the successor of z and x will be y's only non-NIL child or NIL.

Skeleton for Deletion:

- Find y and x.
- If it's **Case 3**, replace z with y.
- Remove y.
- Start fix ups from x depending on the colour of y.